Arquitectura y Organización de Computadores

De los Sistemas Legados a la Computación de Alto Rendimiento

EDUINCA
EDITORIAL DE EDUCACIÓN, INVESTIGACIÓN Y CULTURA ACADÉMICA

Autora:

Mgs. César Andrés Mero Baquerizo
Universidad de Guayaquil
https://orcid.org/0009-0001-1347-4219
cesar.merob@ug.edu.ec

Mgs. Betty Azucena Macas Padilla
Universidad de Guayaquil
https://orcid.org/0009-0006-2317-6086
betty.macasp@ug.edu.ec

Mgs. Juan Carlos Vasco Delgado
Universidad de Guayaquil
https://orcid.org/0000-0003-0587-9758
juan.vascod@ug.edu.ec

Editorial de Educación, Investigación y Cultura Académica

Mayo 2026

EDUINCA

International Publication Technical Data
Title: Arquitectura y organización de computadores. De los sistemas legados a la computación de alto rendimiento
Authors: César Andrés Mero Baquerizo, Betty Azucena Macas Padilla, Juan Carlos Vasco Delgado.
Publisher: Editorial de Educación, Investigación y Cultura Académica
Cover Design: Editorial de Educación, Investigación y Cultura Académica
Format: PDF
Pages: 248
Size: A4 21x29.7cm
System Requeriments: Adobe Acrobat Reader
Acces Mode: World Wide Web
Publication Date: 11/05/2026
ISBN: 978-9907-9519-9-8

DOI: 10.5281/zenodo.20073198

Primera edición, año 2026. Publicado por Editorial de Educación, Investigación y Cultura Académica.

Editorial de Educación, Investigación y Cultura Académica
Primera edición

ISBN 978-9907-9519-9-8

ÍNDICE DE CONTENIDO

PRÓLOGO

La arquitectura y organización de computadores constituye uno de los campos más decisivos para comprender el presente y el futuro de la sociedad digital. Cada servicio en línea, sistema educativo, plataforma de inteligencia artificial, infraestructura científica, aplicación móvil, entorno de nube, sistema embebido o dispositivo inteligente descansa sobre decisiones arquitectónicas que, aunque muchas veces permanecen invisibles para el usuario final, determinan la velocidad, confiabilidad, seguridad, escalabilidad y eficiencia energética de la experiencia computacional. Este libro, titulado Arquitectura y Organización de Computadores: De los Sistemas Legados a la Computación de Alto Rendimiento, nace precisamente de la necesidad de ofrecer una mirada rigurosa, actualizada y formativa sobre esa arquitectura profunda que sostiene los sistemas informáticos contemporáneos.

La obra propone un recorrido que no se limita a describir componentes, sino que interpreta la evolución de la computación como un proceso histórico, técnico y epistemológico. Desde las máquinas mecánicas y los fundamentos de la arquitectura de Von Neumann hasta los paradigmas emergentes de virtualización, edge computing, DMA avanzado y hardware neuromórfico, el texto busca mostrar que la arquitectura computacional no es una colección estática de conceptos, sino un campo dinámico atravesado por tensiones permanentes: velocidad frente a energía, abstracción frente a control, centralización frente a distribución, compatibilidad frente a especialización, rendimiento frente a seguridad, y escalabilidad frente a sostenibilidad.

El lector encontrará en estas páginas un desarrollo progresivo que parte de los fundamentos históricos y conceptuales de la arquitectura computacional, avanza hacia las jerarquías de memoria y las dinámicas de interconexión, profundiza en la complejidad del microprocesador moderno y culmina con el análisis de las arquitecturas de entrada/salida, virtualización y computación distribuida. Esta estructura responde a una intención pedagógica clara: comprender primero los principios que organizan el sistema computacional, luego los mecanismos que permiten mover y preservar datos, posteriormente las estrategias internas que

sostienen la ejecución instruccional, y finalmente las arquitecturas que expanden la computación hacia la nube, la periferia y los sistemas heterogéneos.

Uno de los aportes centrales de este libro es su tratamiento integral de temas que con frecuencia se estudian de manera fragmentada. La memoria no se analiza únicamente como un componente de almacenamiento temporal, sino como una jerarquía estratégica que condiciona algoritmos, rendimiento y confiabilidad. El microprocesador no se presenta como una unidad secuencial simple, sino como una microarquitectura segmentada, especulativa, paralela y energéticamente condicionada. La entrada/salida no se aborda como un tema periférico, sino como el eje donde convergen DMA, redes de alta velocidad, NVMe, aceleradores, virtualización, seguridad y movimiento eficiente de datos. La nube y el edge no se interpretan como simples modelos de servicio, sino como expresiones de una reorganización profunda del hardware, el software y la infraestructura distribuida.

La obra también asume un compromiso con la actualidad científica. Las discusiones incorporan problemas vigentes de la computación moderna: el cuello de botella de Von Neumann, la brecha entre procesamiento y memoria, el costo energético de los centros de datos, la integración de aceleradores, la virtualización asistida por hardware, la seguridad de hipervisores, la no uniformidad del DMA, las rutas directas entre GPU y almacenamiento, el procesamiento en la periferia y la sostenibilidad de la inteligencia artificial. Estos temas permiten al lector comprender que la arquitectura de computadores no pertenece únicamente al pasado fundacional de la informática, sino que se encuentra en el centro de los desafíos tecnológicos de la próxima década.

Este libro está concebido para estudiantes universitarios, docentes, investigadores y profesionales interesados en comprender con profundidad el funcionamiento de los sistemas computacionales modernos. Su estilo busca equilibrar rigor técnico y claridad expositiva, evitando reducir la complejidad de los temas, pero procurando que cada concepto se articule con una explicación amplia, contextualizada y progresiva. La intención no es ofrecer definiciones aisladas, sino construir una comprensión madura de los principios

arquitectónicos que permiten diseñar, evaluar y proyectar sistemas computacionales eficientes.

En un tiempo marcado por inteligencia artificial, Big Data, automatización industrial, servicios cloud, dispositivos inteligentes y computación ubicua, comprender la arquitectura de computadores se vuelve indispensable. No basta con saber utilizar sistemas digitales; es necesario entender las estructuras que los hacen posibles, las limitaciones que enfrentan y las decisiones de diseño que condicionan su rendimiento. Esta obra invita a mirar más allá de la superficie de la tecnología y adentrarse en la lógica material, organizacional y conceptual de la computación contemporánea.

Finalmente, este prólogo reconoce que la arquitectura computacional es, en última instancia, una disciplina de equilibrio. Cada avance técnico trae consigo nuevas posibilidades, pero también nuevos límites. La historia de los computadores demuestra que no existe rendimiento sin costo, abstracción sin mediación, seguridad sin diseño, ni escalabilidad sin organización. Este libro se ofrece como una guía académica para comprender esos equilibrios y para formar una mirada crítica sobre los sistemas que sostienen la vida digital actual.

INTRODUCCIÓN

La computación moderna se ha convertido en una infraestructura esencial para la vida social, científica, educativa, económica e industrial. Desde los servicios más cotidianos hasta los sistemas de inteligencia artificial de mayor escala, todo proceso digital depende de una arquitectura subyacente que organiza el procesamiento, la memoria, la comunicación, el almacenamiento, la entrada/salida y la interacción entre hardware y software. Sin embargo, esta arquitectura suele permanecer oculta bajo capas de aplicaciones, interfaces y servicios que simplifican su uso, pero invisibilizan su complejidad. Comprender la arquitectura y organización de computadores implica, por tanto, estudiar el fundamento material y lógico que permite que una instrucción se ejecute, que un dato se transfiera, que una memoria responda, que un periférico comunique, que una máquina virtual se aísle y que una infraestructura distribuida sostenga millones de operaciones simultáneas.

El presente libro, Arquitectura y Organización de Computadores: De los Sistemas Legados a la Computación de Alto Rendimiento, desarrolla una visión amplia y articulada de ese campo. Su propósito es ofrecer al lector una comprensión progresiva de los principios que han configurado la evolución de los sistemas computacionales, desde sus antecedentes mecánicos y electrónicos hasta los paradigmas actuales de computación paralela, virtualización, nube, edge computing, aceleración especializada e inteligencia artificial energéticamente sostenible. La obra parte de una premisa central: la arquitectura de computadores no puede entenderse como una disciplina cerrada ni como un conjunto de componentes aislados, sino como un sistema de relaciones entre procesamiento, memoria, comunicación, control, energía, seguridad y datos.

El primer capítulo, titulado Epistemología y Fundamentos de la Arquitectura Computacional, establece la base histórica y conceptual del libro. En él se analiza la evolución de las familias de computadores, desde las primeras máquinas de cálculo hasta el paradigma de Von Neumann, cuya influencia sigue presente en la organización de los sistemas modernos. Este capítulo permite comprender que la computación actual no surge de manera espontánea, sino de una secuencia de problemas y soluciones que fueron configurando el modo en que las máquinas

almacenan instrucciones, procesan datos y coordinan operaciones. Asimismo, se examina la diferencia entre arquitectura funcional y organización física, con el fin de distinguir entre lo que el sistema promete al programador y la forma concreta en que el hardware materializa esa promesa.

Dentro de ese mismo capítulo se aborda la taxonomía de Flynn como una herramienta fundamental para comprender el procesamiento paralelo, así como las métricas de rendimiento y la Ley de Amdahl, indispensables para evaluar los límites reales de la aceleración computacional. La discusión crítica se orienta hacia el cuello de botella de Von Neumann, mostrando que la separación entre memoria y procesador, aunque fue decisiva para la consolidación de la computación moderna, se ha convertido en una limitación persistente frente a las demandas de procesamiento masivo de datos. Así, el capítulo inicial no solo presenta fundamentos, sino que problematiza la vigencia de esos fundamentos en un contexto donde la velocidad del cálculo depende cada vez más de la capacidad de mover, ubicar y reutilizar información.

El segundo capítulo, Dinámicas de Interconexión y Jerarquías de Memoria en la Era del Big Data, profundiza en uno de los problemas más importantes de la arquitectura contemporánea: la distancia entre procesamiento y acceso a datos. A medida que los procesadores aumentaron su capacidad de ejecución, la memoria y los sistemas de comunicación interna se convirtieron en factores críticos del rendimiento. Este capítulo examina las estructuras de buses, los protocolos de arbitraje, la jerarquía de memoria desde registros hasta memoria principal semiconductora, y los sistemas de almacenamiento masivo con configuraciones RAID. La memoria se presenta como una arquitectura de compromisos entre velocidad, capacidad, costo, persistencia, energía y confiabilidad.

El análisis de este capítulo permite comprender que el Big Data no solo plantea un desafío de volumen, sino también de movimiento. Los datos deben desplazarse entre registros, cachés, DRAM, almacenamiento persistente, redes, nodos distribuidos y aceleradores. Cada desplazamiento introduce latencia, consumo energético y posibilidades de error. Por ello, la discusión crítica se centra en la brecha entre velocidad de procesamiento y velocidad de acceso a

memoria, destacando cómo esta diferencia condiciona el diseño de algoritmos científicos y sistemas de alta disponibilidad. El capítulo concluye que la gestión eficiente de datos exige pensar de manera integrada memoria, interconexión, almacenamiento, redundancia, recuperación y localidad.

El tercer capítulo, El Microprocesador Moderno: Paralelismo y Segmentación Instruccional, se concentra en la CPU como núcleo de la ejecución lógica y aritmética, pero evita reducirla a una unidad secuencial simple. El microprocesador moderno se analiza como una microarquitectura compleja que combina ciclo de instrucción, segmentación, predicción de saltos, ejecución fuera de orden, renombramiento de registros, unidades funcionales, extensiones vectoriales y control energético. El capítulo explica cómo la técnica de pipelining permite superponer etapas de ejecución para aumentar throughput, aunque introduce hazards estructurales, de datos y de control que exigen mecanismos de detección, forwarding, interlocks, stalls y flushes.

Asimismo, se estudia la comparación entre RISC y CISC desde una mirada contemporánea, superando oposiciones simplistas. Las arquitecturas modernas han difuminado sus fronteras mediante micro-operaciones internas, extensiones especializadas, microarquitecturas híbridas, toolchains avanzados y políticas de eficiencia energética. También se abordan las unidades de control microprogramadas como puente entre la ISA visible y las señales internas del hardware, y se analiza la predicción de saltos junto con la ejecución fuera de orden como estrategias para sostener el paralelismo instruccional en programas secuenciales. La discusión crítica plantea una tensión fundamental: el rendimiento microarquitectónico ya no puede evaluarse sin considerar energía, verificabilidad, área, predictibilidad y adecuación de carga.

El cuarto capítulo, Paradigmas Emergentes: Entrada/Salida, DMA y Virtualización de Hardware, amplía la mirada hacia los mecanismos que conectan el sistema computacional con su entorno. En este capítulo, la entrada/salida deja de ser tratada como un tema periférico y se convierte en un eje central de la organización moderna. Se analiza la gestión de periféricos, el acceso directo a memoria, las optimizaciones de DMA, DDIO, NUDMA, user-level I/O, NVMe, NVMe-over-Fabrics, GPUDirect Storage y las rutas directas

entre aceleradores y almacenamiento. El argumento central es que el rendimiento de los sistemas actuales depende de mover datos por trayectorias cada vez más cortas, seguras, localizadas y adaptativas.

Este capítulo también examina la tensión entre interrupciones de hardware y polling, mostrando que la atención de eventos de E/S debe adaptarse al contexto de carga, latencia y consumo energético. Posteriormente, se aborda el impacto de la virtualización en la abstracción de recursos físicos, incluyendo hipervisores, máquinas virtuales, traducción de memoria, IOMMU, seguridad y virtualización asistida por hardware. Finalmente, se estudian las arquitecturas orientadas a la nube, el edge computing, fog, mist, microservicios, cloud-native, serverless y hardware neuromórfico para centros de datos sostenibles. La discusión crítica plantea que la próxima década exigirá arquitecturas de E/S contextuales, heterogéneas, seguras y energéticamente conscientes.

En conjunto, los cuatro capítulos desarrollan una comprensión integral de la arquitectura computacional desde sus fundamentos hasta sus desafíos emergentes. La obra sostiene que el diseño de sistemas eficientes no depende de una sola dimensión técnica, sino de la articulación entre procesamiento, memoria, comunicación, almacenamiento, virtualización, seguridad, energía y distribución. La arquitectura moderna ya no se define únicamente por la CPU ni por el modelo de Von Neumann, sino por un ecosistema de recursos heterogéneos que deben coordinarse para responder a cargas de trabajo cada vez más complejas.

Esta introducción invita al lector a recorrer el libro como una secuencia de profundización conceptual. Primero, se comprenden los fundamentos; luego, se estudian las rutas de datos; después, se analiza el núcleo microarquitectónico de ejecución; y, finalmente, se examinan los paradigmas que expanden la computación hacia la nube, la periferia y los sistemas virtualizados. El resultado es una obra que busca contribuir a la formación universitaria, al análisis científico y a la comprensión crítica de las tecnologías que sostienen el mundo digital contemporáneo.

CAPÍTULO I: EPISTEMOLOGÍA Y FUNDAMENTOS DE LA ARQUITECTURA COMPUTACIONAL

Chapter I: Epistemology and Fundamentals of Computer Architecture

CAPÍTULO I: EPISTEMOLOGÍA Y FUNDAMENTOS DE LA ARQUITECTURA COMPUTACIONAL

Chapter I: Epistemology and Fundamentals of Computer Architecture

I. INTRODUCCIÓN

La arquitectura computacional constituye uno de los campos más decisivos para comprender la forma en que el conocimiento humano logró transformarse en procedimiento, el procedimiento en algoritmo y el algoritmo en operación física ejecutable por una máquina. Su estudio no puede reducirse a la descripción de procesadores, memorias, buses, registros o unidades de control, porque detrás de cada componente técnico existe una historia intelectual más profunda: la historia de la formalización del cálculo, de la representación simbólica de la información y de la búsqueda de dispositivos capaces de automatizar procesos que antes pertenecían exclusivamente al razonamiento humano. En este sentido, la arquitectura de computadores es, al mismo tiempo, una disciplina técnica y una expresión material de la epistemología computacional, pues permite observar cómo una idea abstracta sobre el procesamiento de información se convierte en una organización funcional, física y operativa. La ciencia de la computación ha sido interpretada como un sistema de conocimiento en evolución, en el cual los conceptos de computación, información y algoritmo se forman, transforman y acumulan históricamente, desde los dispositivos mecánicos de cálculo hasta los sistemas contemporáneos de inteligencia algorítmica (Thinh & Tung, 2026).

Esta perspectiva obliga a situar el origen de la arquitectura computacional mucho antes de la aparición del computador electrónico moderno. Las máquinas mecánicas, los instrumentos de cálculo y los dispositivos aritméticos tempranos no fueron simples antecedentes tecnológicos, sino manifestaciones iniciales de una aspiración epistemológica persistente: hacer que una operación intelectual pudiera ser externalizada, repetida, controlada y verificada mediante un artefacto. Desde el ábaco hasta las máquinas de cálculo más complejas, la humanidad fue desplazando progresivamente el cálculo desde el cuerpo y la memoria hacia sistemas externos de inscripción, representación y manipulación. En esa trayectoria, el cálculo dejó de ser solo una práctica humana asistida por

instrumentos y empezó a convertirse en una estructura formalizable. La computación moderna surge precisamente cuando esa tradición instrumental se articula con la lógica matemática, la teoría de la computabilidad y la ingeniería electrónica, configurando una nueva manera de comprender la relación entre pensamiento, información y máquina.

El tránsito desde la computación mecánica hacia la computación algorítmica no puede entenderse sin la formalización lógica y matemática de los siglos XIX y XX. La consolidación de la lógica simbólica, los sistemas formales, la teoría de funciones computables y la noción de algoritmo permitió que el problema del cálculo adquiriera una profundidad distinta: ya no se trataba únicamente de calcular más rápido, sino de determinar qué podía ser calculado, bajo qué reglas, con qué límites y mediante qué estructura de procedimiento. La reconstrucción histórico-epistemológica de la informática reconoce como momentos decisivos la emergencia de la lógica formal, la computación abstracta y los trabajos fundacionales vinculados a la posibilidad de concebir la computación como proceso algorítmico formalizable (Thinh & Tung, 2026). De esta manera, la arquitectura computacional no aparece como una invención aislada de la ingeniería, sino como la consecuencia material de una larga elaboración conceptual en torno a la posibilidad de ejecutar reglas mediante sistemas físicos.

En este proceso, el paradigma de Von Neumann ocupa una posición estructurante. Su importancia no radica únicamente en haber definido una forma de organizar componentes internos, sino en haber proporcionado una solución general para convertir el algoritmo en proceso material. La arquitectura de programa almacenado permitió que instrucciones y datos residieran en una memoria común, haciendo posible que una misma máquina pudiera ejecutar múltiples tareas mediante la modificación del programa y no mediante la reconstrucción física del dispositivo. Esta idea transformó la naturaleza del computador: de máquina especializada pasó a ser máquina general de procesamiento simbólico. En términos epistemológicos, la arquitectura de Von Neumann permitió que el conocimiento procedimental pudiera codificarse, almacenarse, recuperarse y ejecutarse como una secuencia de instrucciones. Por ello, su influencia se extiende más allá del diseño clásico de computadores, ya que constituye una de las bases técnicas de la informática moderna, la simulación

científica, los sistemas de inteligencia artificial y la computación de propósito general (Thinh & Tung, 2026).

No obstante, la arquitectura de Von Neumann también introdujo una tensión que acompaña a la computación hasta la actualidad. Al separar conceptualmente el procesamiento, la memoria y las operaciones de entrada y salida, el modelo hizo posible una organización clara y flexible del computador, pero también configuró una dependencia crítica entre la unidad de procesamiento y el acceso a memoria. Esta relación, que en un primer momento permitió ordenar funcionalmente la máquina, se convirtió progresivamente en una fuente de limitación frente al crecimiento de la velocidad de los procesadores, la expansión de los datos y la complejidad de las aplicaciones científicas. La arquitectura computacional contemporánea se desarrolla, en buena medida, como una respuesta a esta tensión: mantener la universalidad y la flexibilidad del programa almacenado, pero superar las restricciones impuestas por la latencia, el ancho de banda, la comunicación interna, la jerarquía de memoria y la necesidad de paralelismo.

Desde esta óptica, el computador no debe entenderse como un artefacto neutro que simplemente ejecuta órdenes, sino como un sistema de mediación entre abstracciones formales y procesos físicos. La distinción entre especificación, algoritmo y proceso computacional permite comprender que una instrucción no se agota en su formulación simbólica, puesto que debe ser implementada materialmente en estados físicos de la máquina. Durán (2013) sostiene que el proceso computacional puede concebirse como la concreción física del algoritmo, en la medida en que el computador, estructurado bajo reglas incorporadas en el hardware, transforma una secuencia formal en activación material de compuertas lógicas, memoria, buses y dispositivos de entrada y salida. Esta comprensión resulta central para el presente capítulo, porque permite diferenciar la arquitectura como modelo funcional de la organización como realización física, sin separar artificialmente la teoría del cálculo de la materialidad electrónica que la hace operativa.

La evolución de las familias de computadores expresa, por tanto, una doble transformación. Por un lado, revela la historia de los soportes tecnológicos: engranajes, relés, tubos de vacío, transistores, circuitos integrados,

microprocesadores, sistemas multinúcleo, aceleradores, clusters y plataformas distribuidas. Por otro lado, manifiesta una evolución en la forma de concebir el procesamiento de información: desde la ejecución secuencial de operaciones aritméticas hasta la organización paralela, distribuida y heterogénea del cómputo. El desarrollo del transistor y del microprocesador facilitó la expansión de los computadores hacia múltiples ámbitos de la vida científica, industrial y cotidiana, mientras que la interconexión de máquinas mediante redes abrió el camino hacia nuevas formas de computación distribuida y ciencia intensiva en datos (Vallverdú, 2009). Así, cada familia computacional no solo representa una mejora técnica, sino una reorganización de las posibilidades de cálculo, almacenamiento, comunicación y producción de conocimiento.

En la actualidad, el análisis de la arquitectura computacional se vuelve inseparable de la ciencia intensiva en datos. La investigación científica contemporánea depende de simulaciones, bases de datos masivas, inteligencia artificial, supercomputadores, visualización, middleware, redes distribuidas y recursos de cómputo geográficamente dispersos. Este escenario ha dado lugar a lo que Vallverdú (2009) denomina epistemología computacional, entendida como el estudio de los procesos computacionales implicados en la producción de conocimiento humano. Desde esta perspectiva, los computadores ya no son únicamente herramientas auxiliares para acelerar operaciones, sino infraestructuras cognitivas que participan en la creación, validación, representación y circulación del conocimiento científico. El cambio es profundo: la máquina no solo calcula resultados previamente concebidos por el investigador, sino que también habilita nuevas formas de descubrimiento, modelización, análisis y explicación.

Esta transformación explica por qué la arquitectura computacional debe estudiarse junto con el rendimiento. Un sistema eficiente no se define únicamente por la frecuencia de reloj del procesador ni por la cantidad de instrucciones que puede ejecutar por segundo, sino por la relación compleja entre tiempo de ejecución, latencia, ancho de banda, jerarquía de memoria, paralelismo, consumo energético, escalabilidad y adecuación de la arquitectura a la carga de trabajo. En este punto, las métricas de rendimiento dejan de ser simples indicadores cuantitativos y se convierten en criterios de diseño. Evaluar

un computador implica preguntarse cómo circulan los datos, cómo se distribuyen las instrucciones, qué fracción de un problema puede paralelizarse, cuánto tiempo se pierde en comunicación y sincronización, y qué límites impone la parte secuencial de un proceso. Por ello, la Ley de Amdahl conserva una importancia teórica fundamental: recuerda que el aumento de recursos de procesamiento no garantiza, por sí mismo, una aceleración proporcional, ya que todo sistema paralelo está condicionado por las porciones no paralelizables de la tarea.

La búsqueda de mayor rendimiento condujo al desarrollo de arquitecturas paralelas, vectoriales, multiprocesador, multinúcleo y de alto desempeño. En este contexto, la taxonomía de Flynn continúa siendo un marco clásico para comprender las formas básicas de organización paralela a partir de la relación entre flujos de instrucciones y flujos de datos. La clasificación en SISD, SIMD, MISD y MIMD ha permitido ordenar históricamente distintas arquitecturas de alto rendimiento, aunque también presenta limitaciones frente a sistemas modernos cada vez más heterogéneos y difíciles de clasificar mediante categorías rígidas (Van der Steen, 2008). Su incorporación en este capítulo no tendrá un propósito meramente descriptivo, sino analítico: permitirá mostrar cómo el paralelismo reconfigura la arquitectura computacional al desplazar el foco desde la ejecución secuencial de instrucciones hacia la coordinación de múltiples unidades de procesamiento, memorias, redes de interconexión y modelos de distribución de carga.

El capítulo se organiza a partir de esta problemática general. En primer lugar, se examina la evolución de las familias de computadores como una trayectoria histórico-epistemológica que vincula tecnologías, paradigmas de cálculo y formas de organización del conocimiento. En segundo lugar, se analiza la diferencia entre arquitectura funcional y organización física, destacando la importancia de los niveles de abstracción para comprender cómo una especificación lógica se materializa en circuitos, buses, registros, unidades de control y memoria. En tercer lugar, se aborda la taxonomía de Flynn como una puerta de entrada al procesamiento paralelo y a la comprensión de las arquitecturas de alto rendimiento. En cuarto lugar, se desarrollan las métricas de rendimiento y la Ley de Amdahl como herramientas conceptuales para evaluar los límites reales de la aceleración computacional. Posteriormente, la discusión crítica problematiza el

cuello de botella de Von Neumann frente a las exigencias contemporáneas de procesamiento masivo de datos, inteligencia artificial, simulación científica y sistemas distribuidos.

En conjunto, este capítulo plantea que la arquitectura computacional moderna debe entenderse como el resultado de una tensión permanente entre abstracción y materialidad, universalidad y especialización, secuencialidad y paralelismo, velocidad de procesamiento y acceso a memoria. Su estudio permite comprender no solo cómo están organizados los computadores, sino también por qué ciertas formas de organización han llegado a dominar la historia de la informática y por qué hoy se encuentran sometidas a revisión. La arquitectura de computadores, en consecuencia, no es un campo cerrado ni puramente instrumental; es una zona de convergencia entre lógica, ingeniería, epistemología, ciencia de datos y diseño tecnológico. Desde esta base, el análisis de los sistemas legados y de la computación de alto rendimiento puede abordarse con una mirada crítica, capaz de reconocer tanto la vigencia de los principios clásicos como la necesidad de repensarlos ante las demandas de una sociedad cada vez más dependiente del procesamiento intensivo de información.

II. DESARROLLO

Evolución de las familias de computadores

La evolución de las familias de computadores no debe interpretarse como una sucesión lineal de dispositivos técnicos cada vez más veloces, sino como un proceso histórico de reorganización del conocimiento computacional, en el que se transforman simultáneamente los soportes materiales del cálculo, las formas de representación simbólica y las condiciones de posibilidad del procesamiento automatizado de información. Desde una lectura epistemológica, el computador moderno no aparece de manera repentina con la electrónica digital, sino que es el resultado de una trayectoria en la que el cálculo humano fue progresivamente exteriorizado, formalizado y materializado en artefactos capaces de ejecutar operaciones bajo reglas. En esta perspectiva, la ciencia de la computación puede comprenderse como un sistema de conocimiento en evolución, articulado por etapas que van desde la computación mecánica hasta la inteligencia algorítmica,

atravesando momentos de formalización lógica, teoría computacional y consolidación de sistemas digitales modernos (Thinh & Tung, 2026).

Las primeras formas de cálculo asistido, representadas por instrumentos como el ábaco, las tablas numéricas, los mecanismos de conteo y las máquinas aritméticas tempranas, inauguraron una relación fundamental entre técnica y pensamiento: la posibilidad de trasladar operaciones cognitivas repetitivas hacia dispositivos externos. En estos sistemas, el cálculo todavía permanecía estrechamente vinculado a la manipulación física de cantidades, pero ya se insinuaba una condición que sería central para la arquitectura computacional posterior: la operación podía estabilizarse en una estructura material. Vallverdú (2009) recuerda que, desde culturas sedentarias antiguas, la necesidad de registrar y procesar cantidades condujo tanto al uso de marcas materiales como al diseño de máquinas especiales para efectuar cálculos, entre ellas el ábaco, lo que evidencia que la computación posee raíces históricas asociadas a la memoria, el registro y la administración de información.

Sin embargo, el paso decisivo hacia la computación moderna no consistió únicamente en perfeccionar los mecanismos de cálculo, sino en convertir el cálculo en una estructura formal. La lógica matemática, el álgebra booleana, la teoría de funciones computables y los modelos abstractos de máquina permitieron desplazar el centro del problema desde el artefacto hacia el procedimiento. En ese desplazamiento, la pregunta fundamental dejó de ser solo cómo calcular más rápido y se convirtió en una interrogante más profunda: qué puede ser calculado, bajo qué reglas, con qué límites y mediante qué tipo de representación. La arquitectura computacional heredó esta transformación, porque toda máquina digital contemporánea se sostiene sobre la posibilidad de reducir operaciones, instrucciones y datos a sistemas simbólicos discretos que pueden ser codificados físicamente. En consecuencia, la arquitectura de computadores debe entenderse como la materialización técnica de una teoría previa del cálculo, no como una simple acumulación de componentes electrónicos.

La formalización lógico-matemática también permitió establecer una distinción que resulta esencial para comprender la evolución posterior de las familias

computacionales: la diferencia entre algoritmo, implementación y proceso físico. El algoritmo puede concebirse como una estructura abstracta de pasos definidos, mientras que su ejecución requiere una plataforma material que interprete dichas instrucciones y las convierta en estados físicos. Durán (2013) desarrolla esta relación al diferenciar entre especificación, algoritmo y proceso computacional, señalando que el proceso computacional constituye la concreción física del algoritmo en una máquina organizada bajo reglas incorporadas en el hardware. Esta idea resulta clave para comprender por qué las familias de computadores no son únicamente generaciones tecnológicas, sino formas históricas de implementar materialmente procedimientos formales.

La aparición del computador electrónico de programa almacenado representó una ruptura decisiva dentro de esta trayectoria. Antes de esta configuración, muchas máquinas estaban orientadas a tareas particulares o dependían de modificaciones físicas complejas para alterar su comportamiento. El principio de programa almacenado permitió que instrucciones y datos residieran en una memoria común, haciendo posible que una misma máquina ejecutara diferentes procedimientos mediante cambios en el programa y no en la estructura física principal. Desde el punto de vista arquitectónico, este principio transformó al computador en una máquina de propósito general; desde el punto de vista epistemológico, permitió que el conocimiento procedimental fuera codificado, conservado, recuperado y ejecutado como información manipulable por la propia máquina. Por ello, el paradigma asociado a Von Neumann no solo organizó procesador, memoria y entrada/salida, sino que configuró una forma moderna de comprender la relación entre algoritmo y máquina.

Las primeras familias electrónicas basadas en tubos de vacío deben valorarse dentro de ese contexto. Aunque estos sistemas eran voluminosos, costosos, frágiles y energéticamente demandantes, demostraron que el cálculo automatizado podía superar ampliamente la escala operativa humana. Su relevancia no descansa únicamente en el aumento de velocidad, sino en haber creado una infraestructura material para el cálculo científico, militar, estadístico y administrativo. En estas máquinas, la arquitectura computacional comenzó a consolidarse como un campo de diseño que debía resolver simultáneamente problemas de representación, almacenamiento, control, confiabilidad y

ejecución. La máquina electrónica temprana no era todavía una tecnología de uso masivo, pero sí una plataforma capaz de modificar la escala de los problemas abordables, especialmente en contextos donde la cantidad de operaciones exigía un nivel de automatización imposible para el cálculo manual.

La transición hacia el transistor y los circuitos integrados generó una segunda transformación estructural. La reducción del tamaño físico, el incremento de confiabilidad, la disminución relativa del consumo energético y la posibilidad de integrar múltiples componentes en superficies cada vez menores modificaron el horizonte de la computación. El computador dejó de ser una instalación excepcional y empezó a convertirse en una infraestructura expandible hacia laboratorios, empresas, universidades, industrias y, posteriormente, hogares. Vallverdú (2009) identifica este proceso como parte de una transformación más amplia en la que el desarrollo de las máquinas electrónicas, la revolución del transistor y el microprocesador facilitaron la incorporación de computadores en múltiples situaciones, hasta configurar nuevas formas de vida, investigación y pensamiento mediadas por tecnologías digitales.

El microprocesador marcó una fase decisiva porque integró en un solo circuito funciones que previamente requerían múltiples unidades físicas. Este cambio no solo alteró la escala técnica de fabricación, sino también la escala social de apropiación del cómputo. La familia de los microcomputadores y computadores personales abrió la posibilidad de que la computación dejara de estar concentrada exclusivamente en instituciones científicas, militares o empresariales, y se desplazara hacia ámbitos individuales, educativos, domésticos y profesionales. Con ello, el computador se convirtió en una herramienta de producción simbólica, comunicación, aprendizaje, cálculo, diseño y gestión. No obstante, esta democratización del cómputo no sustituyó a las grandes arquitecturas, sino que coexistió con ellas, generando una diversificación creciente entre computadores personales, servidores, mainframes, sistemas embebidos y plataformas especializadas.

La expansión de las familias computacionales también evidencia que la evolución técnica no consiste en reemplazos absolutos. Los sistemas contemporáneos conservan principios heredados de arquitecturas previas mientras incorporan

nuevas estrategias de organización. Un computador personal moderno sigue dependiendo de memoria, instrucciones, procesamiento y entrada/salida, pero puede integrar múltiples núcleos, memoria caché multinivel, unidades gráficas, controladores especializados y mecanismos de virtualización. De manera similar, los servidores empresariales mantienen principios de programa almacenado, pero se diseñan para disponibilidad, concurrencia, seguridad, redundancia y procesamiento multiusuario. Esta coexistencia de capas históricas demuestra que las familias de computadores se desarrollan por sedimentación y recombinación, no por cancelación completa de lo anterior.

El desarrollo de sistemas de alto rendimiento añadió otra dimensión a esta evolución. Cuando los problemas científicos y técnicos comenzaron a demandar escalas superiores de cálculo, la mejora de un solo procesador dejó de ser suficiente. La arquitectura tuvo que orientarse hacia la coordinación de múltiples unidades de procesamiento, memorias, redes y mecanismos de comunicación. Van der Steen (2008) explica que la arquitectura de los computadores de alto rendimiento debe analizarse considerando clases macroarquitectónicas, procesadores, redes y aceleradores, puesto que el desempeño efectivo no depende únicamente de la CPU, sino también de las formas de interconexión y de las estrategias de organización del sistema.

La computación paralela y distribuida, por tanto, representa una transformación en la forma de concebir el rendimiento. Durante décadas, el aumento de potencia se vinculó con procesadores más rápidos, frecuencias superiores y mejoras internas en la ejecución de instrucciones. Sin embargo, los límites térmicos, energéticos y físicos de esa estrategia impulsaron el tránsito hacia arquitecturas multicore, manycore, vectoriales, gráficas y de clusters. Ngoko y Trystram (2016) advierten que, ante la dificultad de continuar incrementando indefinidamente la potencia de procesadores individuales, una vía dominante para sostener el aumento de capacidad ha sido agrupar varios núcleos o unidades de cómputo dentro de una misma plataforma, reorientando el problema del rendimiento hacia el paralelismo y la coordinación.

Esta reorientación modificó la relación entre algoritmo y arquitectura. En una máquina secuencial, el algoritmo puede representarse principalmente como una

sucesión ordenada de instrucciones. En una arquitectura paralela, en cambio, el problema debe descomponerse, distribuirse, sincronizarse y recomponerse. Esto significa que el diseño arquitectónico interviene de manera directa en la estructura de la solución, porque la eficiencia depende de cómo se reparten los datos, cómo se comunican los procesos, cómo se administra la memoria y cómo se reduce la espera entre unidades de procesamiento. Así, la familia de los sistemas paralelos no se define solamente por tener varios procesadores, sino por introducir una forma distinta de organizar el trabajo computacional.

La evolución reciente también ha desplazado la noción de computador desde la máquina individual hacia la infraestructura distribuida. La computación en la nube transformó los recursos de procesamiento, almacenamiento y software en servicios accesibles mediante redes, apoyados en virtualización, elasticidad y modelos de provisión bajo demanda. Shawish y Salama (2014) plantean que el cloud computing convirtió la antigua aspiración de la computación como utilidad en una realidad operativa, al ofrecer recursos escalables, virtualizados y accesibles a través de internet. Esta familia de sistemas no se define por un dispositivo único, sino por una arquitectura de centros de datos, redes, servicios, acuerdos de nivel de servicio y modelos de consumo flexible. Con ello, el computador se convierte en una infraestructura abstracta que el usuario utiliza sin necesariamente conocer su localización física o su organización interna detallada.

No obstante, la centralización de la nube produjo nuevos desafíos asociados con latencia, tráfico de red, consumo energético, soberanía de datos y capacidad de respuesta en tiempo real. Frente a ello, las arquitecturas de borde y niebla surgieron como alternativas que acercan el procesamiento hacia los usuarios, sensores o fuentes de datos. Li et al. (2018) sostienen que el edge computing aparece como una extensión de los servicios de la nube hacia sistemas más próximos al usuario, con el propósito de reducir latencia, sobrecarga de comunicación y limitaciones derivadas de la centralización de datos. Esta tendencia introduce una nueva familia arquitectónica en la que el cómputo se distribuye espacialmente, obligando a pensar la arquitectura no solo en términos de componentes internos, sino también de localización, conectividad, movilidad y contexto.

Finalmente, las familias de computadores no convencionales amplían el horizonte conceptual de la arquitectura computacional al mostrar que el cómputo no está necesariamente restringido al silicio. La computación molecular, cuántica, neuromórfica o bioinspirada plantea que los procesos de información pueden implementarse en soportes físicos alternativos, siempre que exista una estructura capaz de representar estados, transformar símbolos o producir resultados bajo reglas. Păun, Rozenberg y Salomaa (1998/2003) explican que la computación con ADN se apoya en fenómenos como el paralelismo masivo de cadenas moleculares y la complementariedad Watson-Crick, lo que permite concebir formas de procesamiento diferentes a las arquitecturas electrónicas tradicionales. Aunque estas líneas no sustituyen de inmediato a los computadores digitales convencionales, sí muestran que la arquitectura computacional debe ser comprendida como el estudio de las formas materiales del procesamiento, no como la descripción exclusiva de una tecnología dominante.

La evolución de las familias de computadores revela, en consecuencia, una tensión permanente entre continuidad y ruptura. Existe continuidad porque muchos sistemas actuales conservan principios de programa almacenado, representación binaria, memoria jerárquica y ejecución de instrucciones. Pero también existe ruptura porque las demandas contemporáneas han impulsado arquitecturas paralelas, distribuidas, heterogéneas y especializadas que ya no pueden describirse plenamente desde el modelo secuencial clásico. El computador moderno es, cada vez más, una articulación de unidades funcionales diversas, memorias de distintos niveles, redes de comunicación, dispositivos inteligentes, servicios virtualizados y aceleradores orientados a dominios específicos. Esta complejidad obliga a comprender las familias computacionales como configuraciones históricas que responden a problemas técnicos, científicos y sociales determinados.

La siguiente tabla sintetiza esta evolución desde una perspectiva histórico-epistemológica. Su finalidad es organizar las principales familias computacionales no como una cronología cerrada, sino como una secuencia de transformaciones en los soportes, formas de procesamiento, aportes arquitectónicos y limitaciones que impulsaron nuevas soluciones.

Tabla 1

Evolución epistemológica y tecnológica de las familias de computadores

Etapa o familia computacional	Soporte tecnológico dominante	Forma principal de procesamiento	Aporte epistemológico y arquitectónico	Limitación que impulsa la transición
Dispositivos mecánicos de cálculo	Engranajes, ruedas, mecanismos físicos y dispositivos aritméticos	Automatización parcial de operaciones numéricas	Exteriorización inicial del cálculo humano en artefactos reproducibles	Rigidez funcional, baja velocidad y escasa reprogramabilidad
Computación formal y lógica	Sistemas simbólicos, álgebra booleana, modelos matemáticos de computación	Manipulación abstracta de símbolos y reglas	Definición teórica de algoritmo, computabilidad y límites formales del cálculo	Falta de implementación electrónica general y escalable
Primeros computadores electrónicos	Tubos de vacío, relés avanzados, memorias tempranas	Ejecución automática de operaciones digitales	Consolidación del computador como máquina de cálculo científico e institucional	Alto consumo, gran tamaño, baja confiabilidad relativa y programación compleja
Computadores de programa almacenado	Memoria común para instrucciones y datos, unidad de control y unidad aritmético-lógica	Ejecución secuencial de instrucciones almacenadas	Nacimiento del computador de propósito general bajo el paradigma de Von Neumann	Dependencia crítica entre CPU y memoria central
Computadores transistorizados y de circuitos integrados	Transistores, circuitos integrados y sistemas de menor escala física	Procesamiento digital más confiable y compacto	Expansión del cómputo hacia investigación, industria y administración	Necesidad de mayor integración, menor costo y mayor accesibilidad
Microcomputadores y computadores personales	Microprocesadores, memorias semiconductoras y periféricos de usuario	Procesamiento individual, interactivo y multipropósito	Democratización del acceso al cómputo y expansión social del software	Capacidad limitada frente a problemas científicos o masivos
Servidores, mainframes modernos y sistemas empresariales	Multiprocesadores, almacenamiento robusto, virtualización y redes	Procesamiento transaccional, multiusuario y de alta disponibilidad	Consolidación del cómputo como infraestructura organizacional	Escalabilidad condicionada por costos, memoria, interconexión y administración
Supercomputadores y HPC	Clusters, multinúcleo, GPU, aceleradores y redes de alta velocidad	Paralelismo masivo, simulación y procesamiento científico intensivo	Reconfiguración del rendimiento como problema de coordinación arquitectónica	Complejidad de programación, comunicación, energía y escalabilidad real

Computación en la nube y distribuida	Centros de datos virtualizados, redes globales, almacenamiento remoto	Recursos bajo demanda y procesamiento distribuido	Transformación del computador en servicio e infraestructura flexible	Latencia, centralización, privacidad, dependencia de conectividad y tráfico
Edge, fog y paradigmas posnube	Dispositivos cercanos al usuario, nodos de borde, sensores y gateways	Procesamiento distribuido próximo a la fuente de datos	Redefinición espacial del cómputo y reducción de latencia	Heterogeneidad, gestión de recursos, seguridad y coordinación
Paradigmas no convencionales	ADN, sistemas cuánticos, neuromórficos o biomoleculares	Procesamiento molecular, probabilístico, bioinspirado o especializado	Ampliación del concepto de máquina computacional más allá del silicio	Madurez tecnológica limitada y desafíos de control, error e implementación

Nota. *Elaboración propia a partir del análisis del acervo bibliográfico del capítulo, especialmente Thinh y Tung (2026), Vallverdú (2009), Durán (2013), Van der Steen (2008), Ngoko y Trystram (2016), Shawish y Salama (2014), Li et al. (2018), y Păun, Rozenberg y Salomaa (1998/2003).*

La lectura de la tabla permite advertir que cada familia computacional resolvió un conjunto de problemas y, al mismo tiempo, generó nuevas limitaciones. Las máquinas mecánicas ayudaron a estabilizar el cálculo, pero carecían de flexibilidad; los computadores electrónicos aumentaron la velocidad, pero enfrentaron problemas de tamaño, consumo y confiabilidad; el microprocesador expandió el acceso, pero no bastó para responder a las exigencias del cómputo científico masivo; los sistemas paralelos incrementaron la capacidad de procesamiento, pero hicieron más compleja la programación; la nube ofreció elasticidad, pero profundizó los desafíos de latencia y centralización; y las arquitecturas de borde acercaron el procesamiento a los datos, aunque multiplicaron la heterogeneidad y las dificultades de coordinación. Por ello, la evolución de las familias de computadores debe entenderse como una historia de soluciones parciales ante problemas cambiantes, no como una progresión simple hacia una máquina definitiva.

Esta mirada permite sostener que el estudio de las familias computacionales cumple una función fundacional dentro del capítulo. A partir de él se comprende por qué resulta necesario distinguir entre arquitectura funcional y organización física, por qué la clasificación de Flynn se volvió relevante para analizar el paralelismo, y por qué las métricas de rendimiento son indispensables para

evaluar sistemas modernos. Cada uno de esos temas surge de la misma pregunta histórica: cómo organizar materialmente el procesamiento de información para hacerlo más flexible, confiable, eficiente y escalable. Así, la evolución de las familias de computadores no solo introduce el pasado de la arquitectura computacional, sino que prepara el análisis de sus fundamentos conceptuales y de sus límites contemporáneos.

Arquitectura funcional vs. organización física

La arquitectura computacional exige diferenciar con precisión dos planos que suelen aparecer entrelazados en la operación real de una máquina: la arquitectura funcional y la organización física. La primera se refiere al modelo lógico mediante el cual el sistema define qué operaciones puede realizar, qué instrucciones reconoce, cómo representa los datos, qué registros son visibles, qué modos de direccionamiento admite y qué comportamiento ofrece al programador o al sistema operativo. La segunda corresponde a la forma concreta en que ese modelo se implementa mediante circuitos, unidades funcionales, buses, memoria, rutas de datos, mecanismos de control, jerarquías de almacenamiento y dispositivos de entrada y salida. Esta distinción es importante porque permite comprender que una misma arquitectura funcional puede materializarse en organizaciones físicas diferentes, y que esas diferencias inciden directamente en el rendimiento, el consumo energético, la latencia, la escalabilidad y la eficiencia del sistema.

Desde una perspectiva epistemológica, la arquitectura funcional se sitúa en un nivel intermedio entre el algoritmo abstracto y la máquina material. El algoritmo define una secuencia formal de operaciones; la arquitectura funcional establece el repertorio de acciones que una máquina puede ejecutar; y la organización física convierte esas acciones en procesos eléctricos, lógicos y temporales. Durán (2013) permite fundamentar esta distinción al explicar que el software puede analizarse mediante tres unidades: especificación, algoritmo y proceso computacional. La especificación contiene la información sobre el sistema que se desea modelar y sobre la arquitectura computacional; el algoritmo organiza esa información como procedimiento formal; y el proceso computacional constituye la realización física del algoritmo en la máquina. Esta relación muestra que el computador no es

únicamente un soporte pasivo, sino el dominio material donde una estructura sintáctica se interpreta y se ejecuta mediante estados físicos.

La arquitectura funcional, por tanto, puede entenderse como la interfaz conceptual entre lo que el programa requiere y lo que la máquina está preparada para ejecutar. En este nivel se ubica la arquitectura del conjunto de instrucciones, conocida como ISA, que define las operaciones disponibles para el procesador, los formatos de instrucción, los registros accesibles, los tipos de datos, los mecanismos de direccionamiento y ciertas convenciones de ejecución. Van der Steen (2008) define la Instruction Set Architecture como el conjunto de instrucciones que una CPU está diseñada para ejecutar, destacando que procesadores de fabricantes distintos pueden compartir una misma ISA aunque posean estructuras internas diferentes. Esta observación resulta esencial para diferenciar arquitectura y organización: dos procesadores pueden ejecutar el mismo repertorio de instrucciones y, sin embargo, diferir profundamente en cachés, pipelines, unidades funcionales, predicción de saltos, cantidad de registros físicos, buses internos o estrategias de ejecución.

La organización física, en cambio, remite a las decisiones de implementación que hacen posible el comportamiento definido en el nivel funcional. Incluye la disposición interna del procesador, el diseño de la unidad aritmético-lógica, la cantidad y tipo de registros, el sistema de control, la estructura de la memoria caché, los caminos de datos, las unidades de carga y almacenamiento, los mecanismos de interconexión y las señales que coordinan el movimiento de información. En los materiales sobre taxonomía y organización básica del computador se identifican componentes como el contador de programa, el registro de acceso a memoria, la memoria, el registro de datos de memoria, el acumulador, la unidad aritmético-lógica, el registro de instrucción, el código de operación, el campo de dirección y la unidad lógica de control, los cuales permiten observar cómo una instrucción abstracta se transforma en una secuencia de transferencias internas entre registros, memoria y unidad de procesamiento (Stanford, s. f.).

Esta diferencia entre arquitectura funcional y organización física puede ilustrarse mediante la ejecución de una instrucción simple de carga. Funcionalmente, una

instrucción como LOAD indica que un dato debe trasladarse desde una ubicación de memoria hacia un registro o acumulador. Sin embargo, físicamente esa operación requiere varios pasos coordinados: el contador de programa proporciona la dirección de la instrucción, la memoria entrega su contenido al registro de datos, el registro de instrucción conserva el código, el decodificador identifica la operación, el registro de dirección de memoria apunta al operando, la memoria entrega el dato y finalmente este se transfiere al acumulador o al registro correspondiente. La instrucción parece elemental en el nivel funcional, pero en el nivel organizacional involucra sincronización, señales de control, acceso a memoria, transferencia interna y activación de circuitos. Esa distancia entre simplicidad funcional y complejidad física es una de las claves para comprender la arquitectura computacional.

La unidad de control cumple un papel central en esta articulación. Desde el punto de vista funcional, la máquina parece ejecutar instrucciones de manera ordenada según el programa. Desde el punto de vista físico, esa ejecución requiere interpretar códigos de operación, activar señales, seleccionar registros, coordinar accesos a memoria, definir rutas de datos y asegurar que cada etapa ocurra en el momento adecuado. La unidad de control es, por tanto, el mecanismo que traduce la secuencia lógica del programa en acciones físicas distribuidas dentro del procesador. Su existencia muestra que la arquitectura computacional no se limita a disponer componentes, sino que necesita una gramática temporal de ejecución. Un computador no solo almacena datos e instrucciones; también organiza cuándo, dónde y cómo deben moverse.

La memoria representa otro punto donde la distinción entre función y organización se vuelve decisiva. Funcionalmente, la memoria puede describirse como un espacio direccionable donde se almacenan datos e instrucciones. Organizacionalmente, sin embargo, la memoria es una jerarquía compuesta por registros, cachés, memoria principal, almacenamiento secundario y, en sistemas modernos, memorias distribuidas o compartidas. Cada nivel posee diferentes tiempos de acceso, capacidades, costos y proximidad respecto al procesador. La arquitectura funcional puede presentar la memoria como un espacio coherente de direcciones, pero la organización física debe resolver cómo acercar los datos a las unidades de ejecución, cómo reducir esperas, cómo mantener coherencia y

cómo evitar que la diferencia de velocidad entre procesador y memoria degrade el rendimiento.

La interconexión interna también permite apreciar esta diferencia. Un bus puede entenderse funcionalmente como un medio para transportar datos, direcciones o señales de control entre componentes. Pero, desde la organización física, su ancho, frecuencia, arbitraje, topología, contención y relación con los dispositivos determinan la eficiencia real del sistema. La arquitectura funcional presupone que los datos pueden moverse de una unidad a otra; la organización física debe garantizar que ese movimiento ocurra con un costo temporal y energético aceptable. Esta distinción adquiere especial importancia en computadores de alto rendimiento, donde el problema no consiste únicamente en ejecutar operaciones aritméticas rápidamente, sino en alimentar de datos a las unidades funcionales sin que estas permanezcan inactivas por falta de operandos.

La organización física también introduce restricciones que no siempre son visibles desde el nivel funcional. Un programa puede estar correctamente formulado y una arquitectura puede ofrecer las instrucciones necesarias para ejecutarlo, pero el rendimiento real dependerá de la profundidad del pipeline, la eficiencia de la caché, la predicción de saltos, la disponibilidad de unidades funcionales, el ancho de banda de memoria, la latencia de interconexión y la capacidad del compilador para explotar la estructura del procesador. Van der Steen (2008) muestra este punto al analizar procesadores donde bloques como unidades de carga y almacenamiento, registros enteros y de punto flotante, cachés de instrucciones y datos, unidades de bifurcación, colas y mecanismos de despacho condicionan el rendimiento efectivo más allá del conjunto de instrucciones visible.

Por esta razón, la arquitectura funcional puede permanecer estable mientras la organización física evoluciona de manera significativa. La compatibilidad entre generaciones de procesadores suele depender de conservar la misma ISA o una extensión compatible, mientras que la mejora del rendimiento proviene de reorganizar internamente la máquina. Una familia de procesadores puede mantener instrucciones conocidas para preservar software existente, pero modificar profundamente el número de unidades de ejecución, la jerarquía de

caché, los mecanismos de paralelismo a nivel de instrucción, las técnicas de predicción o la administración energética. Esta situación confirma que la arquitectura funcional define el contrato lógico de la máquina, mientras que la organización física define las estrategias materiales para cumplir ese contrato con mayor o menor eficiencia.

La distinción adquiere todavía mayor importancia en sistemas embebidos y arquitecturas orientadas a dominios específicos. En estos casos, el diseño funcional puede optimizarse para un tipo de aplicación, pero la organización física debe equilibrar flexibilidad, costo, área de silicio, consumo energético y rendimiento. Ştefan (2006) sostiene que, en la computación embebida, la arquitectura del conjunto de instrucciones puede optimizarse para un dominio de aplicación y seguir siendo general dentro de ese dominio, mientras que el flujo de procesamiento y acceso a datos suele ser más predecible que en computadores personales, lo que permite simplificar ciertos mecanismos de caché y atender con mayor cuidado los cuellos de botella de entrada y salida. Esta observación muestra que la arquitectura funcional no se diseña en abstracto, sino en relación con las cargas de trabajo previstas y con las restricciones físicas de implementación.

En los computadores modernos, la organización física tiende a incorporar mecanismos que no alteran necesariamente el modelo funcional visible, pero sí modifican profundamente la ejecución real. La segmentación o pipelining, por ejemplo, permite superponer etapas de varias instrucciones para incrementar el flujo de ejecución. Desde el punto de vista del programador, la máquina sigue ejecutando instrucciones de acuerdo con el programa; desde el punto de vista físico, distintas instrucciones pueden encontrarse simultáneamente en fases de búsqueda, decodificación, ejecución, acceso a memoria o escritura de resultados. Los materiales del acervo describen el pipelining como una técnica de implementación en la que múltiples instrucciones se solapan durante la ejecución, aprovechando el paralelismo existente entre las acciones necesarias para completar una instrucción (Stanford, s. f.).

Este ejemplo permite comprender que la arquitectura funcional tiende a preservar una ilusión de secuencialidad, mientras que la organización física

puede explotar paralelismo interno. El programa puede estar escrito como una secuencia ordenada, pero el procesador puede reorganizar, solapar o anticipar ciertas operaciones siempre que conserve el resultado observable previsto por la arquitectura. Esta diferencia es crucial para comprender microarquitecturas modernas, donde técnicas como ejecución fuera de orden, predicción de saltos, renombramiento de registros, ejecución especulativa y paralelismo interno buscan incrementar el rendimiento sin romper el contrato funcional que el procesador mantiene con el software.

La relación entre arquitectura funcional y organización física también puede analizarse desde la noción de implementación semántica. Durán (2013) argumenta que el proceso computacional representa la materialización del algoritmo en el computador, puesto que las instrucciones se interpretan en un dominio físico compuesto por estados de máquina. Esta idea permite sostener que la organización física no es un simple detalle técnico, sino la condición material que permite que la estructura formal del algoritmo tenga efectos causales en el mundo físico. En otras palabras, el algoritmo no "corre" en abstracto; se ejecuta porque existe una organización de compuertas, registros, buses, memoria y control capaz de convertir instrucciones en transiciones de estado.

Esta comprensión evita dos reduccionismos. El primero consistiría en pensar que el computador es solamente hardware, ignorando que su funcionamiento depende de modelos formales, repertorios de instrucciones, convenciones de representación y capas de abstracción. El segundo consistiría en pensar que la computación es únicamente algoritmo, olvidando que toda ejecución real necesita un soporte físico con límites temporales, energéticos y estructurales. La arquitectura funcional y la organización física no son dimensiones separadas, sino niveles interdependientes. La primera define posibilidades lógicas; la segunda determina condiciones reales de ejecución. Un diseño computacional maduro debe articular ambas dimensiones para que el sistema no solo sea correcto desde el punto de vista formal, sino eficiente, confiable y viable desde el punto de vista físico.

En este sentido, la distinción entre arquitectura funcional y organización física también ayuda a comprender el rendimiento. Cuando un sistema presenta bajo desempeño, la causa puede ubicarse en diferentes niveles: una ISA poco adecuada para cierto tipo de operación, una jerarquía de memoria insuficiente, una unidad funcional saturada, un bus congestionado, un algoritmo con mala localidad de referencia, una mala distribución de datos o una política de compilación ineficiente. Sin la distinción entre niveles, el análisis del rendimiento se vuelve impreciso. Por el contrario, al separar el contrato funcional de la implementación física, es posible identificar si el problema pertenece al modelo de instrucciones, al diseño microarquitectónico, a la organización de memoria, a la comunicación interna o a la interacción con el software.

Esta distinción prepara el análisis de los apartados siguientes. La taxonomía de Flynn clasifica formas de procesamiento según flujos de instrucciones y datos, lo cual se vincula con la arquitectura funcional de los sistemas paralelos, pero también con su organización física en procesadores vectoriales, multiprocesadores, arreglos de procesamiento y sistemas distribuidos. Del mismo modo, las métricas de rendimiento y la Ley de Amdahl dependen de comprender qué parte del comportamiento pertenece al algoritmo, qué parte corresponde al modelo funcional de la máquina y qué parte se explica por la implementación física. La arquitectura computacional, entendida rigurosamente, se sitúa precisamente en esa intersección entre abstracción, organización, ejecución y rendimiento.

La taxonomía de Flynn en el procesamiento paralelo

La taxonomía de Flynn constituye uno de los marcos clásicos más influyentes para comprender las formas básicas de organización del procesamiento paralelo, porque propone clasificar los sistemas computacionales a partir de dos dimensiones fundamentales: el flujo de instrucciones y el flujo de datos. Esta clasificación resulta especialmente relevante dentro de la arquitectura computacional porque permite pasar de una visión centrada en el procesador individual hacia una comprensión más amplia de cómo las máquinas pueden ejecutar operaciones sobre uno o múltiples conjuntos de datos, bajo uno o múltiples flujos de control. Flynn (1972) propuso esta clasificación para analizar

la efectividad de distintas organizaciones computacionales, estableciendo una matriz conceptual que permitió distinguir entre SISD, SIMD, MISD y MIMD como formas generales de relación entre instrucciones y datos. Ngoko y Trystram (2016) destacan que la taxonomía de Flynn sirvió históricamente como una estructura clara para pensar el paralelismo, al organizar las arquitecturas según la multiplicidad de flujos de instrucciones y de datos.

La importancia de esta taxonomía no reside únicamente en su valor clasificatorio, sino en su capacidad para hacer visible una transformación profunda de la arquitectura computacional: el tránsito desde la ejecución secuencial hacia la explotación sistemática del paralelismo. En los sistemas clásicos de inspiración Von Neumann, el procesamiento se concibe principalmente como la ejecución de una secuencia de instrucciones sobre datos almacenados en memoria. Esta forma de organización resulta adecuada para describir el modelo SISD, en el que una única unidad de control administra un flujo de instrucciones que opera sobre un único flujo de datos. Sin embargo, a medida que las demandas de rendimiento aumentaron y los límites físicos de la aceleración de un solo procesador se hicieron más evidentes, la arquitectura computacional tuvo que explorar modos de organizar simultaneidad, concurrencia y distribución del trabajo. En este escenario, la taxonomía de Flynn permite introducir el paralelismo como una propiedad arquitectónica que no se reduce al número de procesadores, sino a la relación entre control, datos y ejecución.

El modelo SISD, o Single Instruction Single Data, representa la forma convencional de procesamiento secuencial. En esta categoría, una máquina ejecuta un solo flujo de instrucciones sobre un solo flujo de datos, siguiendo una lógica de operación ordenada. El modelo se asocia con el computador clásico de una unidad central de procesamiento, en el cual las instrucciones se recuperan, decodifican y ejecutan de manera secuencial. Aunque los procesadores modernos incorporan técnicas internas como segmentación, caché, predicción de saltos o ejecución especulativa, el modelo SISD conserva valor conceptual porque expresa la forma más básica de relación entre programa y datos. Los materiales de arquitectura revisados en el acervo presentan el SISD como el sistema de una sola instrucción y un solo dato, vinculado con máquinas secuenciales y con el modelo de pila en ciertos diseños históricos (Stanford, s. f.).

No obstante, el SISD no debe entenderse como una categoría obsoleta. Su importancia radica en que proporciona el punto de referencia desde el cual se evalúa el paralelismo. La mayor parte de las métricas de aceleración comparan el desempeño paralelo contra un tiempo de ejecución secuencial, lo que convierte al modelo SISD en una base conceptual para medir ganancias, límites y desviaciones. Además, muchos sistemas contemporáneos ejecutan secciones de código que, por dependencias de datos, control o memoria, permanecen esencialmente secuenciales. Por ello, aun en arquitecturas multicore o distribuidas, la lógica SISD subsiste como una dimensión interna de ciertos procesos. La arquitectura paralela no elimina la secuencialidad, sino que intenta reducir su peso relativo dentro del tiempo total de ejecución.

El modelo SIMD, o Single Instruction Multiple Data, introduce una transformación significativa: una misma instrucción se aplica simultáneamente a múltiples datos. Esta forma de organización resulta particularmente eficiente cuando se trabaja con arreglos, vectores, matrices, imágenes, señales o conjuntos de datos homogéneos sobre los cuales se debe ejecutar la misma operación de manera repetida. En términos arquitectónicos, el SIMD explota la regularidad de los datos y reduce el costo de control, porque una sola instrucción gobierna muchas operaciones elementales. Esta lógica ha sido fundamental en procesadores vectoriales, unidades multimedia, GPU y aceleradores utilizados en cómputo científico e inteligencia artificial. Los materiales del acervo describen los sistemas SIMD como aquellos en los que una instrucción opera sobre múltiples datos, y asocian esta categoría con procesadores vectoriales y arreglos de procesamiento (Stanford, s. f.).

La fuerza del SIMD se encuentra en su correspondencia con problemas que poseen alto grado de paralelismo de datos. Si una operación debe repetirse sobre miles o millones de elementos independientes, resulta ineficiente ejecutar cada operación mediante instrucciones separadas. En cambio, una organización SIMD permite que la máquina aproveche la estructura regular del problema para incrementar el rendimiento. Esta lógica explica la relevancia actual de las GPU, cuya eficiencia proviene de ejecutar muchas operaciones similares sobre grandes volúmenes de datos. Aunque las GPU modernas son más complejas que el modelo SIMD clásico, conservan una afinidad conceptual con esta categoría debido a su

orientación hacia el paralelismo masivo de datos. Ştefan (2006) sostiene que el paralelismo de datos constituye una forma natural de computación paralela, especialmente cuando múltiples elementos programables ejecutan instrucciones idénticas o similares sobre datos diferenciados.

El modelo MISD, o Multiple Instruction Single Data, es el más problemático de la taxonomía de Flynn. En términos formales, supone que múltiples flujos de instrucciones actúan sobre un único flujo de datos. Esta definición resulta conceptualmente posible, pero históricamente ha tenido menos realizaciones prácticas que SISD, SIMD y MIMD. La dificultad radica en imaginar sistemas donde varias instrucciones diferentes procesen el mismo dato de manera estructuralmente útil y general. Algunos enfoques han asociado MISD con arquitecturas de tolerancia a fallos, sistemas redundantes, procesamiento en pipeline o arreglos sistólicos, aunque estas asociaciones han sido debatidas. Los materiales didácticos del acervo advierten que algunas interpretaciones ubican el pipelining dentro de MISD, pero reconocen que esta clasificación es discutible (Stanford, s. f.).

La categoría MISD, pese a sus dificultades, posee valor epistemológico porque obliga a pensar formas de paralelismo que no se ajustan cómodamente al paralelismo de datos ni al multiprocesamiento convencional. Ngoko y Trystram (2016) señalan que el MISD ha sido históricamente una clase poco comprendida, y proponen revisitarla a partir de enfoques de portafolio de algoritmos, donde múltiples procedimientos concurren sobre una misma instancia del problema para encontrar una solución de manera cooperativa. Esta interpretación desplaza la comprensión del paralelismo desde el nivel estricto de la instrucción hacia el nivel de algoritmos concurrentes, lo que permite reconsiderar MISD como una estrategia de cooperación entre métodos diferentes que actúan sobre un mismo problema.

El modelo MIMD, o Multiple Instruction Multiple Data, representa una de las categorías más relevantes para la computación contemporánea. En esta organización, múltiples flujos de instrucciones operan sobre múltiples flujos de datos, lo que permite ejecutar tareas independientes, subtareas relacionadas o procesos cooperativos dentro de una misma carga de trabajo. Esta categoría

incluye multiprocesadores, multicomputadores, clusters, sistemas multinúcleo y muchas arquitecturas de alto rendimiento. Su relevancia se explica porque la mayoría de los problemas complejos no pueden resolverse únicamente aplicando la misma instrucción a múltiples datos, sino que requieren descomponer el trabajo en procesos diferenciados, cada uno con su propio flujo de control y datos. Flynn (1972) proporcionó el marco inicial para distinguir esta forma de organización frente a las demás clases, y su utilidad se mantiene porque permite reconocer la complejidad inherente de los sistemas con múltiples unidades de ejecución.

La arquitectura MIMD introduce desafíos distintos a los del SIMD. En SIMD, el problema central consiste en alimentar eficientemente a muchas unidades que ejecutan una misma operación. En MIMD, en cambio, deben coordinarse flujos de ejecución potencialmente diferentes, administrar dependencias, sincronizar procesos, distribuir memoria, evitar condiciones de carrera y equilibrar carga. Esta complejidad aumenta cuando el sistema utiliza memoria distribuida, porque los datos pueden encontrarse en distintos nodos y la comunicación se convierte en un factor decisivo del rendimiento. Las arquitecturas MIMD, por tanto, no solo requieren múltiples procesadores, sino mecanismos de coordinación capaces de sostener coherencia, comunicación y sincronización. En este punto, la taxonomía de Flynn ofrece una entrada conceptual, pero no basta para describir todas las variaciones internas de los sistemas modernos.

La siguiente tabla sintetiza las cuatro categorías de la taxonomía de Flynn y su relevancia dentro del análisis contemporáneo de la arquitectura computacional. Su propósito es organizar la relación entre flujos de instrucciones, flujos de datos, ejemplos conceptuales y desafíos arquitectónicos, evitando una lectura meramente memorística de las siglas.

Tabla 2

Taxonomía de Flynn y formas de paralelismo computacional

Categoría	Flujo de instrucciones	Flujo de datos	Forma de procesamiento	Ejemplo arquitectónico o conceptual	Relevancia y desafío actual
SISD	Único	Único	Ejecución secuencial de instrucciones	Procesador clásico de propósito general	Sigue siendo base conceptual del modelo

			sobre datos individuales		secuencial y referencia para medir aceleración paralela
SIMD	Único	Múltiple	Una misma operación se aplica simultáneamente a varios datos	Procesadores vectoriales, GPU, arreglos de procesamiento	Alta relevancia en gráficos, IA, simulación y cálculo matricial; depende de regularidad de datos
MISD	Múltiple	Único	Diferentes instrucciones actúan sobre un mismo flujo de datos	Sistemas redundantes, interpretaciones debatidas de pipeline, portafolios algorítmicos	Categoría menos convencional; útil para repensar cooperación, tolerancia a fallos y modelos algorítmicos concurrentes
MIMD	Múltiple	Múltiple	Diversos procesos ejecutan instrucciones diferentes sobre datos diferentes	Multicore, multiprocesadores, clusters, sistemas distribuidos	Dominante en HPC y sistemas modernos; exige coordinación, sincronización, comunicación y balance de carga

Nota. *Elaboración propia a partir de Flynn (1972), Stanford (s. f.), Ştefan (2006), y Ngoko y Trystram (2016).*

La taxonomía de Flynn permite observar que el paralelismo no es una categoría homogénea. No es lo mismo paralelizar datos que paralelizar instrucciones, ni coordinar múltiples núcleos que ejecutar una operación vectorial. Tampoco es equivalente aumentar unidades de procesamiento que lograr eficiencia efectiva. Un sistema puede poseer muchos procesadores y, sin embargo, obtener bajo rendimiento si las dependencias, la comunicación o la memoria impiden que esos recursos trabajen de manera coordinada. Esta distinción resulta crucial porque evita asociar paralelismo con simple multiplicación de hardware. La arquitectura paralela exige identificar la estructura del problema, el tipo de datos, las dependencias internas, la forma de comunicación y el modelo de programación más adecuado.

A pesar de su valor histórico, la taxonomía de Flynn presenta limitaciones frente a las arquitecturas contemporáneas. Los sistemas actuales combinan con frecuencia múltiples formas de paralelismo dentro de una misma plataforma. Un procesador moderno puede ejecutar instrucciones de manera segmentada, incluir unidades vectoriales SIMD, operar dentro de un sistema multicore MIMD, conectarse a una GPU con paralelismo masivo de datos y formar parte de un cluster distribuido. En estos casos, clasificar la máquina bajo una sola categoría resulta insuficiente. Ştefan (2006) argumenta que la taxonomía de Flynn fue útil durante la era del macroparalelismo, pero se vuelve restrictiva en implementaciones SoC y aplicaciones embebidas que requieren más de una forma de paralelismo funcional.

Esta limitación no invalida la taxonomía, sino que obliga a utilizarla como marco de entrada y no como explicación exhaustiva. Su utilidad principal reside en ofrecer un lenguaje inicial para diferenciar formas de relación entre instrucciones y datos. Sin embargo, la complejidad actual requiere complementarla con criterios de memoria, interconexión, granularidad, sincronización, modelo de programación, heterogeneidad y eficiencia energética. La arquitectura contemporánea ya no puede analizarse únicamente por el número de flujos de instrucciones y datos, porque sus decisiones de diseño dependen también de la proximidad de la memoria, el ancho de banda disponible, el consumo por operación, la especialización de aceleradores y la capacidad del software para explotar los recursos disponibles.

La revisión crítica de la taxonomía de Flynn permite, además, conectar este subtema con la problemática general del capítulo. El paradigma clásico de Von Neumann ofrece una base secuencial que fue extraordinariamente exitosa para la computación de propósito general, pero las exigencias de procesamiento masivo han desplazado el centro del diseño hacia configuraciones paralelas y heterogéneas. En ese desplazamiento, SISD representa la raíz secuencial, SIMD expresa la explotación del paralelismo de datos, MIMD simboliza la organización multiproceso y distribuida, mientras que MISD funciona como una categoría conceptual que invita a explorar modelos menos convencionales de cooperación. El valor de Flynn (1972) consiste precisamente en haber propuesto una estructura que todavía permite leer las tensiones entre control, datos y rendimiento, aunque

hoy deba ser ampliada para abarcar arquitecturas que combinan múltiples estrategias de ejecución.

En consecuencia, la taxonomía de Flynn debe asumirse como un instrumento analítico de transición. Permite pasar del estudio de la organización física básica del computador hacia la comprensión de sistemas capaces de procesar grandes volúmenes de información mediante paralelismo. También prepara el terreno para discutir las métricas de rendimiento y la Ley de Amdahl, porque toda forma de paralelismo debe evaluarse no solo por su estructura, sino por su capacidad real de reducir el tiempo de ejecución. El problema central ya no es únicamente clasificar una arquitectura, sino determinar cuánta aceleración puede obtenerse, qué parte del trabajo permanece secuencial, qué costos introduce la comunicación y qué eficiencia se alcanza al aumentar los recursos de procesamiento.

Métricas de rendimiento y la Ley de Amdahl

El análisis de la arquitectura computacional no puede completarse sin examinar las métricas que permiten evaluar el rendimiento real de un sistema. Si la evolución de las familias de computadores muestra cómo la máquina ha cambiado históricamente, y la distinción entre arquitectura funcional y organización física permite comprender cómo se articulan abstracción e implementación, las métricas de rendimiento permiten responder una pregunta decisiva: hasta qué punto una arquitectura ejecuta de manera eficiente las tareas para las que fue diseñada. En este sentido, el rendimiento no debe reducirse a la frecuencia de reloj, al número de núcleos, a la cantidad de memoria o al volumen de operaciones por segundo. Tales indicadores pueden ser relevantes, pero solo adquieren significado cuando se relacionan con la carga de trabajo, la organización de memoria, la comunicación entre componentes, el modelo de programación, la eficiencia energética y la proporción de trabajo que realmente puede beneficiarse del paralelismo.

La idea de rendimiento computacional ha evolucionado junto con las propias arquitecturas. En los computadores secuenciales clásicos, la evaluación tendía a centrarse en el tiempo necesario para ejecutar un programa o en la cantidad de

instrucciones completadas por unidad de tiempo. En sistemas paralelos y distribuidos, en cambio, el rendimiento se vuelve multidimensional, porque la ejecución depende de la coordinación entre recursos. Un sistema con muchos procesadores puede ser menos eficiente que otro con menos unidades si la comunicación, la sincronización, la latencia de memoria o el acceso a datos generan esperas prolongadas. Por ello, medir rendimiento implica analizar no solo cuánto calcula un sistema, sino cómo distribuye el trabajo, cómo mueve los datos, cómo administra la memoria y cómo convierte recursos disponibles en resultados efectivos. Gunarathne, Wu, Qiu y Fox (2010) muestran esta complejidad al comparar aplicaciones biomédicas paralelas en entornos cloud y MapReduce, evaluando no solo desempeño, sino también eficiencia, costo y usabilidad en función de la naturaleza de la computación ejecutada.

Una primera métrica fundamental es el tiempo de ejecución, entendido como el intervalo requerido para completar una tarea específica bajo condiciones determinadas. Aunque parece una medida directa, su interpretación exige precaución. El tiempo de ejecución puede variar según el tamaño de entrada, el tipo de datos, la arquitectura utilizada, el sistema operativo, el compilador, la ubicación de los archivos, la memoria disponible, la contención de recursos y el tráfico de red. En sistemas de alto rendimiento, el tiempo de ejecución también depende de la forma en que se distribuyen los procesos, de la granularidad de las tareas y de la eficiencia de la comunicación. Por ello, no basta con afirmar que una arquitectura es más rápida; es necesario indicar bajo qué carga, con qué configuración, usando qué número de recursos y frente a qué línea base secuencial o paralela se realiza la comparación.

Otra métrica relevante es el throughput, o capacidad de procesamiento por unidad de tiempo. Mientras el tiempo de ejecución se concentra en cuánto tarda una tarea particular en completarse, el throughput analiza cuántas tareas, transacciones, solicitudes u operaciones puede procesar el sistema en un intervalo determinado. Esta métrica es especialmente importante en servidores, sistemas empresariales, plataformas en la nube, centros de datos, redes de servicios y aplicaciones de procesamiento masivo. Un sistema puede tener una latencia individual moderada, pero un throughput elevado si es capaz de atender muchas solicitudes simultáneamente. A la inversa, puede responder muy rápido

a una tarea aislada y, sin embargo, degradarse cuando aumenta la concurrencia. Esta diferencia obliga a no confundir rapidez individual con capacidad agregada de servicio.

La latencia constituye una tercera métrica esencial. Se refiere al tiempo de espera asociado a una operación, comunicación, acceso a memoria o respuesta del sistema. En arquitectura computacional, la latencia aparece en múltiples niveles: acceso a registros, caché, memoria principal, almacenamiento secundario, redes internas, interconexión entre nodos, entrada y salida, y comunicación con servicios remotos. Su importancia crece en arquitecturas distribuidas, sistemas de borde, aplicaciones interactivas, realidad aumentada, vehículos autónomos, sensores, inteligencia artificial en tiempo real y servicios dependientes de respuesta inmediata. Li, Xue, Wang, Zhang y Li (2018) señalan que los paradigmas orientados al borde surgen, entre otras razones, por las limitaciones de la centralización en la nube, especialmente la latencia, el tráfico de datos y la sobrecarga de comunicación cuando las aplicaciones requieren análisis y control en tiempo real.

El ancho de banda complementa el análisis de latencia. Mientras la latencia mide el tiempo de respuesta o demora inicial, el ancho de banda expresa la cantidad de datos que pueden transferirse por unidad de tiempo. En un sistema de memoria, red o almacenamiento, ambos factores interactúan, pero no son equivalentes. Una memoria puede tener alto ancho de banda y, aun así, presentar latencias relevantes para ciertos patrones de acceso. Del mismo modo, una red puede transmitir grandes volúmenes de datos, pero no ser adecuada para tareas que requieren respuestas inmediatas. Esta distinción es clave para comprender por qué algunas aplicaciones científicas, gráficas o de inteligencia artificial se benefician de arquitecturas con gran ancho de banda, mientras que aplicaciones interactivas o de control en tiempo real dependen de latencias reducidas.

La eficiencia paralela constituye una métrica especialmente importante cuando se evalúan arquitecturas multicore, clusters, GPU, sistemas distribuidos o plataformas de alto rendimiento. Su propósito es determinar qué tan bien se aprovechan los recursos adicionales incorporados al sistema. En términos generales, una ejecución paralela ideal esperaría que duplicar los recursos

redujera el tiempo de ejecución aproximadamente a la mitad. Sin embargo, en la práctica esto rara vez ocurre, porque la coordinación entre recursos introduce costos adicionales. Gunarathne et al. (2010) definen la eficiencia paralela como una medida que relaciona el mejor tiempo secuencial con el tiempo paralelo y el número de núcleos empleados, pero advierten que esta métrica no permite comparar directamente tecnologías distintas si no se consideran diferencias de memoria, ancho de banda, red e infraestructura subyacente.

Esta advertencia es fundamental para evitar interpretaciones simplistas del paralelismo. Un sistema puede mostrar una eficiencia razonable en una aplicación altamente paralelizable y un desempeño deficiente en otra con mayor dependencia de memoria, comunicación o sincronización. De igual modo, una plataforma puede parecer superior en términos de tiempo total, pero resultar menos conveniente si su costo, consumo energético o complejidad de programación son excesivos. Por ello, la eficiencia debe ser leída dentro de una ecología de métricas. En arquitectura computacional, el rendimiento efectivo es una relación entre cálculo, comunicación, memoria, energía, costo, facilidad de programación y naturaleza del problema.

El speedup, o aceleración, permite expresar cuánto mejora el tiempo de ejecución de una tarea al pasar de una ejecución secuencial a una paralela o de una arquitectura base a una arquitectura optimizada. Conceptualmente, se calcula como la razón entre el tiempo de referencia y el tiempo obtenido con la mejora incorporada. Si una tarea tarda 100 unidades de tiempo en una ejecución secuencial y 25 en una ejecución paralela, el speedup es 4. Esta métrica es poderosa porque permite observar el impacto relativo de una optimización, pero también puede resultar engañosa si se presenta sin contexto. No indica por sí sola cuántos recursos se usaron, qué costos se añadieron, si la mejora es sostenible al aumentar el tamaño del problema o si el sistema opera cerca de sus límites de eficiencia.

La escalabilidad se relaciona con la capacidad de un sistema para mantener o mejorar su rendimiento cuando aumentan los recursos, el tamaño del problema o la cantidad de usuarios. Una arquitectura escalable no es simplemente aquella que permite añadir procesadores, memoria o nodos, sino aquella que puede

convertir ese crecimiento en mayor capacidad útil sin que los costos de coordinación superen las ganancias. En sistemas distribuidos, la escalabilidad depende de la topología de red, la gestión de tareas, la distribución de datos y la tolerancia a fallos. En cloud y edge computing, depende también de la asignación dinámica de recursos, de la proximidad a los datos y de la coordinación entre capas. Zhou, Zhang y Xiong (2017) explican que los paradigmas posnube surgen precisamente porque la computación centralizada en la nube encuentra dificultades para atender aplicaciones ubicuas, móviles y sensibles a la latencia, lo que exige nuevas formas de distribuir infraestructura computacional más cerca de los usuarios y dispositivos.

La métrica de costo también resulta ineludible, especialmente en entornos cloud, HPC institucional y sistemas empresariales. Dos arquitecturas pueden ofrecer tiempos de ejecución similares, pero diferir ampliamente en costo de adquisición, mantenimiento, energía, licenciamiento, personal especializado o pago por uso. En la computación como servicio, el análisis de rendimiento se vincula con el modelo económico de la infraestructura. Gunarathne et al. (2010) muestran que diferentes tipos de instancias en la nube pueden ofrecer relaciones distintas entre tiempo de cómputo y costo, y que la plataforma más rápida no siempre coincide con la más económica para una aplicación determinada. Esta observación resulta relevante porque desplaza la evaluación desde el rendimiento absoluto hacia la adecuación entre arquitectura, carga de trabajo y restricciones institucionales.

La Ley de Amdahl ocupa un lugar central dentro de esta discusión porque establece un límite teórico a la aceleración que puede obtenerse mediante paralelización. Su principio básico indica que la mejora global de un sistema está restringida por la fracción de la tarea que no puede beneficiarse de la optimización. Aplicada al paralelismo, la ley sostiene que el speedup máximo depende de la proporción paralelizable del programa y de la fracción que debe ejecutarse de manera secuencial. Amdahl (1967) formuló esta idea en el contexto de los límites prácticos del procesamiento paralelo, mostrando que incluso un número muy grande de procesadores no produce aceleración ilimitada si una parte del programa conserva dependencia secuencial.

La expresión clásica de la Ley de Amdahl puede representarse de la siguiente manera:

$$S(n) - \frac{1}{(1-P) + \frac{P}{n}}$$

En esta fórmula, S(n) representa la aceleración obtenida con n procesadores, mientras que P corresponde a la fracción paralelizable del programa. La parte (1–P) representa la fracción secuencial que no se beneficia de agregar procesadores. El valor de la fórmula revela una conclusión decisiva: cuando n crece, el término $\frac{P}{n}$ disminuye, pero la fracción secuencial permanece. Por ello, la aceleración máxima tiende a estar dominada por aquello que no puede paralelizarse. Esta idea introduce un principio de sobriedad arquitectónica: no todo problema mejora proporcionalmente con más hardware, y no toda inversión en paralelismo produce ganancias equivalentes.

La Ley de Amdahl obliga a pensar el rendimiento desde la estructura interna del problema. Si una tarea es altamente paralelizable, el aumento de procesadores puede producir mejoras importantes. Si, en cambio, posee dependencias secuenciales fuertes, el speedup se limita rápidamente. Este principio permite explicar por qué algunas aplicaciones de simulación, procesamiento gráfico, álgebra lineal, aprendizaje profundo o análisis masivo de datos pueden beneficiarse de GPU y clusters, mientras que otras tareas con dependencias de control, acceso irregular a memoria o alta sincronización no escalan de la misma manera. En consecuencia, el paralelismo no es una solución universal, sino una estrategia condicionada por la naturaleza algorítmica y organizacional del problema.

La relevancia de Amdahl no se limita a una fórmula matemática. Su aporte principal consiste en mostrar que el rendimiento arquitectónico debe evaluarse en términos sistémicos. Mejorar una parte del sistema no garantiza una mejora proporcional del todo. Un procesador más rápido puede quedar limitado por memoria lenta; una GPU potente puede permanecer subutilizada si los datos no llegan con suficiente velocidad; un cluster grande puede perder eficiencia por comunicación excesiva; una aplicación distribuida puede degradarse por latencia

de red; y una arquitectura de borde puede ahorrar transmisión hacia la nube, pero enfrentar restricciones locales de energía y capacidad. La Ley de Amdahl enseña que el sistema completo se comporta según sus cuellos de botella, no según su componente más rápido.

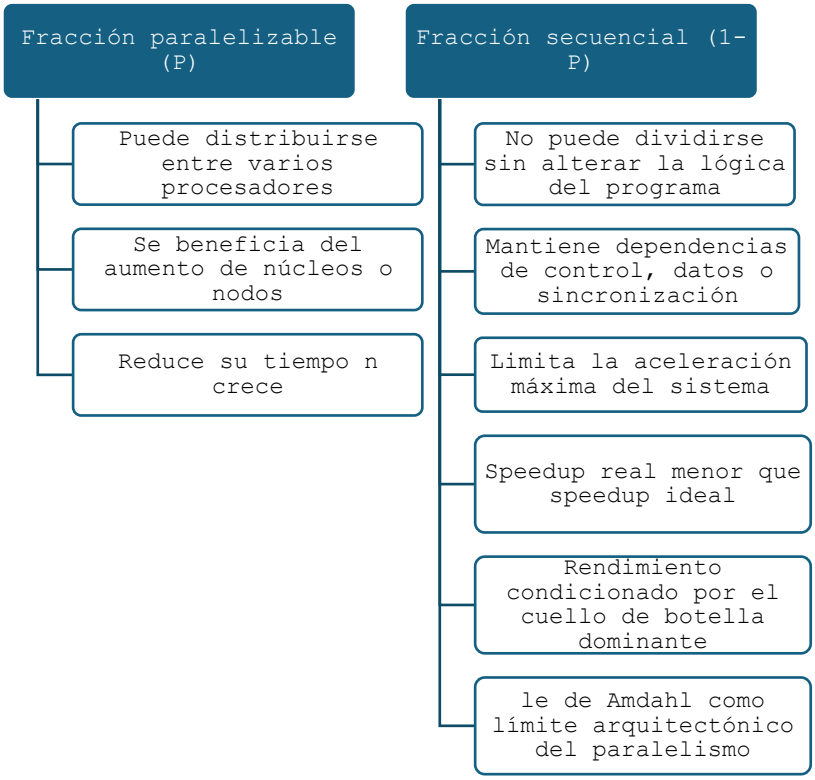
La Ley de Amdahl también permite comprender por qué el diseño arquitectónico debe orientarse a la optimización de los casos dominantes. Si una pequeña parte del programa consume la mayor parte del tiempo de ejecución, optimizar esa sección puede producir más beneficio que mejorar componentes poco utilizados. Si una operación es frecuente, conviene acelerarla; si un acceso a memoria domina el tiempo total, conviene mejorar localidad, caché o ancho de banda; si una comunicación entre nodos genera esperas, conviene rediseñar la distribución de tareas. En este sentido, la ley no solo limita expectativas, sino que orienta decisiones de diseño: identificar dónde se concentra el tiempo, dónde se produce espera y qué parte del sistema condiciona el resultado global.

Por ello, las métricas de rendimiento no deben analizarse como indicadores aislados, sino como instrumentos para comprender la relación entre arquitectura y problema. El tiempo de ejecución muestra el resultado global; el throughput revela capacidad agregada; la latencia evidencia tiempos de espera; el ancho de banda señala capacidad de transferencia; el speedup mide mejora relativa; la eficiencia paralela evalúa uso de recursos; la escalabilidad analiza comportamiento ante crecimiento; el consumo energético introduce sostenibilidad; y el costo conecta el rendimiento con la viabilidad económica. La Ley de Amdahl articula estas métricas recordando que toda mejora está condicionada por la fracción que permanece limitada. Así, el rendimiento computacional no es una propiedad simple de la máquina, sino una relación compleja entre carga de trabajo, arquitectura funcional, organización física, software, comunicación, memoria y contexto de uso.

Para visualizar la relación conceptual entre paralelismo, aceleración y límite estructural, resulta pertinente representar la Ley de Amdahl como una relación entre dos zonas del programa: la porción que puede distribuirse entre varios recursos y la porción que permanece necesariamente secuencial. Esta representación permite observar que el rendimiento paralelo no depende solo del

número de procesadores, sino de la proporción efectiva de trabajo susceptible de paralelización. De este modo, el gráfico no debe leerse como una simplificación matemática, sino como una síntesis arquitectónica del problema: la máquina puede multiplicar recursos, pero el programa conserva dependencias, esperas y segmentos que restringen la mejora global.

Figura 1. ***Relación conceptual entre paralelismo, speedup y límite de Amdahl***



Nota. *Elaboración propia a partir del principio de aceleración limitada propuesto por Amdahl (1967), aplicado al análisis de arquitecturas paralelas y sistemas de alto rendimiento.*

La figura muestra que la fracción paralelizable de un programa puede reducir su tiempo de ejecución conforme aumenta el número de procesadores, núcleos o unidades de cómputo disponibles. Sin embargo, la fracción secuencial permanece como una zona resistente a la distribución del trabajo. Esta zona puede estar asociada a dependencias lógicas, operaciones de inicialización, secciones críticas, sincronización entre procesos, comunicación de resultados, acceso compartido a memoria o pasos que deben ejecutarse en orden. Por ello, el speedup ideal, que

supondría una reducción proporcional del tiempo al aumentar recursos, rara vez coincide con el speedup real. La diferencia entre ambos expresa precisamente la distancia entre la disponibilidad material de hardware y la posibilidad algorítmica de aprovecharlo.

La Ley de Amdahl permite comprender que el rendimiento no es una propiedad acumulativa simple. Agregar procesadores no equivale automáticamente a multiplicar la velocidad del sistema, porque toda arquitectura paralela enfrenta costos de coordinación. La comunicación entre procesos consume tiempo; la sincronización introduce esperas; la distribución de datos puede producir desequilibrios; las jerarquías de memoria pueden generar accesos no uniformes; y las tareas con dependencias fuertes reducen la posibilidad de ejecución concurrente. En este sentido, la aceleración paralela es siempre una relación entre oportunidad y restricción. La oportunidad está dada por la fracción del problema que puede ejecutarse simultáneamente; la restricción está determinada por todo aquello que permanece secuencial, compartido, dependiente o difícil de distribuir.

Esta idea resulta especialmente relevante para interpretar la diferencia entre rendimiento teórico y rendimiento efectivo. Los fabricantes de hardware suelen expresar la capacidad máxima de un sistema mediante cifras de operaciones por segundo, número de núcleos, frecuencia, ancho de banda o capacidad de memoria. Sin embargo, el rendimiento obtenido por una aplicación concreta puede ser significativamente menor si el programa no logra explotar esa capacidad. Amdahl (1967) permite fundamentar esta diferencia al mostrar que la mejora global queda acotada por la parte no optimizada o no paralelizable del proceso. Por ello, en arquitectura computacional no basta con diseñar máquinas potentes; es necesario diseñar sistemas equilibrados, capaces de reducir cuellos de botella y de adecuarse a la estructura real de las cargas de trabajo.

La evaluación del rendimiento exige, además, diferenciar entre escalabilidad fuerte y escalabilidad débil. La escalabilidad fuerte analiza cómo disminuye el tiempo de ejecución de un problema fijo cuando aumentan los recursos. En este caso, la Ley de Amdahl tiene una presencia muy clara, porque la fracción secuencial se vuelve cada vez más dominante a medida que se agregan

procesadores. La escalabilidad débil, en cambio, observa cómo se comporta el sistema cuando el tamaño del problema crece junto con la cantidad de recursos. Esta distinción es importante porque muchos sistemas de alto rendimiento no se diseñan únicamente para resolver el mismo problema más rápido, sino para resolver problemas más grandes, con más datos, más variables, mayor resolución o más simulaciones simultáneas. La evaluación arquitectónica debe considerar ambas formas de escalamiento, ya que cada una revela límites diferentes del sistema.

En sistemas científicos, la escalabilidad está estrechamente vinculada con la estructura del modelo computacional. Las simulaciones físicas, el análisis de genomas, el procesamiento de imágenes, los modelos climáticos, los sistemas de inteligencia artificial y las aplicaciones de minería de datos poseen patrones de paralelismo distintos. Algunas cargas pueden dividirse en tareas casi independientes, mientras que otras requieren comunicación permanente entre procesos. Las primeras tienden a escalar con mayor facilidad, especialmente cuando los datos pueden fragmentarse sin dependencias fuertes. Las segundas enfrentan límites más severos, porque cada avance parcial depende de información producida por otras partes del sistema. De ahí que el rendimiento de una arquitectura no pueda evaluarse de manera universal; siempre debe vincularse con el tipo de problema que se desea resolver.

La noción de localidad también resulta fundamental dentro de las métricas de rendimiento. Un programa eficiente no solo realiza pocas operaciones, sino que accede a los datos de manera favorable para la jerarquía de memoria. La localidad temporal permite reutilizar datos recientemente accedidos; la localidad espacial permite aprovechar datos cercanos en memoria. Cuando un algoritmo posee buena localidad, las cachés pueden reducir de manera significativa la latencia de acceso. Cuando la localidad es deficiente, el procesador puede pasar gran parte del tiempo esperando datos, aunque disponga de unidades de ejecución rápidas. Este fenómeno confirma que el rendimiento no se define únicamente por la capacidad de cálculo, sino por la relación entre cálculo y movimiento de datos. En arquitecturas modernas, mover datos puede ser tan costoso o más costoso que operar sobre ellos.

La relación entre rendimiento y memoria anticipa la discusión crítica sobre el cuello de botella de Von Neumann. La arquitectura clásica separa procesamiento y almacenamiento, lo que obliga a transferir continuamente instrucciones y datos entre memoria y CPU. Mientras la velocidad de los procesadores creció de forma acelerada, la memoria y los canales de comunicación no siempre acompañaron ese crecimiento al mismo ritmo. Esta brecha produjo una situación en la que las unidades de procesamiento pueden estar disponibles, pero subutilizadas por falta de datos oportunos. El problema se agrava en cargas intensivas, donde no basta con realizar cálculos rápidos, sino que es necesario alimentar de manera constante a múltiples unidades de ejecución. En consecuencia, muchas métricas de rendimiento terminan revelando el mismo problema arquitectónico: la eficiencia depende de reducir las esperas impuestas por memoria, comunicación y sincronización.

En este marco, la eficiencia energética se convierte en un criterio de evaluación inseparable del rendimiento. La arquitectura contemporánea ya no puede perseguir únicamente el máximo número de operaciones por segundo, porque el costo energético de mover datos, mantener centros de datos, refrigerar sistemas, alimentar aceleradores y sostener infraestructuras distribuidas condiciona la viabilidad técnica y económica del cómputo. Un sistema puede ser extremadamente rápido, pero inviable si su consumo energético resulta desproporcionado frente a la tarea que ejecuta. Por ello, los indicadores modernos tienden a considerar relaciones como rendimiento por watt, operaciones por joule o costo energético por tarea. Estas métricas revelan que la eficiencia no consiste solo en terminar antes, sino en hacerlo con un uso racional de recursos materiales.

El costo de programación también debe ser incorporado al análisis, aunque no siempre aparezca en métricas tradicionales de hardware. Una arquitectura puede ofrecer gran potencial paralelo, pero si exige modelos de programación excesivamente complejos, su aprovechamiento real se reduce. El programador debe comprender la distribución de memoria, las dependencias, los mecanismos de sincronización, los patrones de comunicación, las bibliotecas disponibles y las restricciones de la plataforma. En sistemas heterogéneos, esta complejidad se incrementa porque una misma aplicación puede distribuir partes de su ejecución

entre CPU, GPU, aceleradores, memoria local, memoria compartida y almacenamiento remoto. El rendimiento efectivo, por tanto, depende también de la capacidad del ecosistema de software para expresar paralelismo sin introducir errores, sobrecostos o ineficiencias.

Las métricas de rendimiento permiten reconocer que la arquitectura computacional opera bajo una lógica de compromiso. Optimizar una dimensión puede afectar otra. Reducir latencia puede incrementar costo; aumentar throughput puede elevar consumo energético; ampliar memoria puede modificar tiempos de acceso; agregar procesadores puede aumentar comunicación; especializar hardware puede reducir flexibilidad; distribuir el cómputo puede mejorar escalabilidad, pero introducir problemas de consistencia y coordinación. El diseño arquitectónico consiste, en gran medida, en administrar estas tensiones. Por ello, no existe una arquitectura universalmente óptima, sino configuraciones más o menos adecuadas para dominios, cargas y restricciones específicas.

Esta perspectiva resulta esencial para comprender la transición hacia arquitecturas heterogéneas. En lugar de buscar una sola unidad general capaz de resolver todos los problemas con eficiencia equivalente, los sistemas modernos combinan componentes especializados: CPU para control y tareas generales, GPU para paralelismo de datos, aceleradores para operaciones de inteligencia artificial, memorias jerárquicas para reducir latencia, redes de alta velocidad para comunicación y almacenamiento distribuido para grandes volúmenes de información. La medición del rendimiento en estos sistemas exige observar la interacción entre componentes. Un acelerador puede ser muy eficiente en cálculo, pero su beneficio se reduce si el traslado de datos desde la memoria principal consume demasiado tiempo. Una red rápida puede mejorar comunicación entre nodos, pero no resolver desequilibrios de carga. Una caché amplia puede reducir accesos a memoria, pero no corregir algoritmos con patrones irregulares de datos.

La Ley de Amdahl también ayuda a comprender por qué el diseño de sistemas eficientes requiere identificar el recurso dominante en cada escenario. En algunas aplicaciones, el límite está en la CPU; en otras, en la memoria; en otras, en el

almacenamiento; en otras, en la red; y en otras, en la sincronización entre procesos. Cuando el recurso limitante cambia, también debe cambiar la estrategia de optimización. Un programa limitado por cálculo puede beneficiarse de más unidades aritméticas; uno limitado por memoria puede requerir mejor localidad, caché o ancho de banda; uno limitado por comunicación puede necesitar una redistribución de tareas; uno limitado por entrada y salida puede exigir almacenamiento más rápido o procesamiento más cercano a los datos. Así, la evaluación del rendimiento se convierte en una práctica diagnóstica que orienta el rediseño arquitectónico y algorítmico.

En consecuencia, las métricas de rendimiento no deben considerarse únicamente como instrumentos de medición posterior, sino como principios de diseño. Antes de construir o seleccionar una arquitectura, es necesario preguntarse qué tipo de carga ejecutará, qué proporción de trabajo es paralelizable, qué volumen de datos debe moverse, qué latencia tolera la aplicación, qué nivel de disponibilidad requiere, qué restricciones energéticas existen y qué costo económico puede asumirse. Esta lógica es especialmente importante en educación computacional, porque evita que el aprendizaje de arquitectura se reduzca a memorizar componentes. Comprender rendimiento implica aprender a pensar la máquina como un sistema de relaciones, límites y compromisos.

Desde esta mirada, la Ley de Amdahl conserva una vigencia conceptual notable. Aunque fue formulada en un contexto anterior a muchas arquitecturas actuales, su principio sigue siendo aplicable porque todo sistema posee partes que escalan y partes que no escalan. La forma específica de la fracción secuencial puede cambiar: en un procesador clásico puede ser una instrucción dependiente; en un cluster puede ser comunicación; en una GPU puede ser transferencia de memoria; en la nube puede ser latencia de red; en el borde puede ser restricción energética; en inteligencia artificial puede ser acceso a datos o sincronización de parámetros. Pero la lógica general se mantiene: la mejora global está limitada por el componente o la etapa menos escalable.

El análisis de las métricas de rendimiento y de la Ley de Amdahl permite cerrar el desarrollo conceptual del capítulo con una idea central: la arquitectura computacional no se evalúa por su potencia aislada, sino por su capacidad para

convertir recursos en ejecución útil. Esta conversión depende del equilibrio entre organización física, modelo funcional, algoritmo, datos y contexto. El rendimiento real surge cuando estos elementos se alinean; se degrada cuando alguno de ellos impone esperas, dependencias o sobrecostos. Por tanto, el diseño de sistemas eficientes requiere comprender tanto la estructura interna de la máquina como la naturaleza del problema que se desea resolver. Esta conclusión prepara la discusión crítica del capítulo, centrada en las limitaciones del cuello de botella de Von Neumann frente a las demandas actuales de procesamiento masivo de datos.

Discusión crítica

La arquitectura de Von Neumann constituye uno de los pilares más influyentes de la computación moderna, no solo porque permitió organizar de manera funcional la relación entre procesador, memoria, instrucciones y datos, sino porque hizo posible la existencia del computador de propósito general como máquina reprogramable. Sin embargo, su éxito histórico no debe impedir una lectura crítica de sus limitaciones. El mismo principio que permitió almacenar instrucciones y datos en una memoria común introdujo una dependencia estructural entre la unidad central de procesamiento y el sistema de memoria. Esta dependencia, conocida como cuello de botella de Von Neumann, se ha convertido en una de las tensiones más persistentes de la arquitectura computacional: el procesador puede incrementar su capacidad de cálculo, pero si los datos y las instrucciones no llegan con suficiente rapidez, la potencia disponible se convierte en capacidad ociosa. En consecuencia, el rendimiento efectivo de un sistema no depende únicamente de la velocidad de cómputo, sino de la capacidad de sostener un flujo equilibrado entre procesamiento, memoria, interconexión y entrada/salida.

Esta tensión resulta especialmente visible en el contexto actual, donde los sistemas computacionales ya no ejecutan únicamente programas secuenciales de tamaño moderado, sino cargas intensivas en datos, simulaciones científicas, modelos de inteligencia artificial, servicios distribuidos, procesamiento multimedia, análisis biomédico, aplicaciones en tiempo real y sistemas ciberfísicos. La arquitectura clásica fue concebida bajo un modelo en el que la

secuencia de instrucciones ocupaba el centro de la ejecución; en cambio, la computación contemporánea se encuentra cada vez más condicionada por el volumen, la localización, la movilidad y la velocidad de acceso a los datos. La transición hacia aplicaciones científicas de gran escala ha mostrado que los problemas de procesamiento no pueden resolverse únicamente con más ciclos de CPU, porque muchas cargas requieren mover, transformar, sincronizar y almacenar información a ritmos que sobrepasan la lógica de una memoria centralizada. En aplicaciones paralelas biomédicas, por ejemplo, el desempeño no depende solo del número de recursos de cómputo, sino también de la eficiencia, el costo, la usabilidad y la adecuación de la plataforma al tipo de datos procesados (Gunarathne et al., 2010).

El cuello de botella de Von Neumann revela, por tanto, una contradicción interna de la computación moderna. Por una parte, la separación entre memoria y procesador permitió flexibilidad, generalidad y reprogramabilidad. Por otra, esa misma separación introdujo una distancia física y lógica entre donde se almacenan los datos y donde se ejecutan las operaciones. Mientras las aplicaciones eran relativamente pequeñas y los procesadores mantenían velocidades moderadas, esta distancia podía gestionarse mediante buses, registros y memoria principal. Pero a medida que los procesadores se hicieron más rápidos, las memorias más jerárquicas y los datos más voluminosos, la diferencia entre capacidad de cálculo y capacidad de alimentación de datos se volvió crítica. La arquitectura respondió con cachés, predicción, segmentación, paralelismo, memoria multinivel, unidades vectoriales y aceleradores; sin embargo, estas soluciones no eliminan el problema de fondo, sino que lo administran mediante capas de compensación.

La Ley de Amdahl permite profundizar esta crítica porque muestra que el rendimiento de un sistema siempre queda condicionado por la parte que no escala. En el caso del cuello de botella de Von Neumann, esa parte no escalable puede estar en la fracción secuencial del programa, en el acceso a memoria, en la comunicación entre procesos, en la sincronización o en la entrada/salida. Agregar procesadores, núcleos o aceleradores puede mejorar la ejecución de ciertas porciones del trabajo, pero no garantiza una mejora proporcional si el flujo de datos permanece restringido. Esta observación cuestiona una visión ingenua del

progreso arquitectónico basada en la simple multiplicación de recursos. La arquitectura contemporánea no enfrenta únicamente el desafío de calcular más, sino de calcular mejor, con menos espera, menor transferencia innecesaria, mayor localidad de datos y una organización más coherente entre hardware y software.

El paralelismo, aunque indispensable, tampoco resuelve por sí solo las limitaciones del modelo clásico. Las arquitecturas SIMD, MIMD, multicore, GPU y de clusters permitieron superar parcialmente la dependencia del procesador único, pero también introdujeron nuevos problemas de coordinación. En los sistemas paralelos, el rendimiento depende de la distribución adecuada de tareas, del balance de carga, de la sincronización, de la comunicación entre unidades y de la coherencia en el acceso a memoria. Ştefan (2006) plantea que las aplicaciones embebidas y de alto rendimiento requieren más de una forma de paralelismo funcional, puesto que la explotación aislada del paralelismo de datos, de instrucciones o de especulación no siempre basta para aproximarse a la eficiencia de soluciones especializadas. Esta afirmación permite cuestionar la idea de que el futuro de la arquitectura dependa de una sola estrategia técnica. La eficiencia contemporánea exige integrar diferentes formas de paralelismo, pero también reconocer que cada una produce costos específicos.

La crítica al paradigma de Von Neumann también debe considerar la tensión entre generalidad y especialización. El computador de propósito general fue una de las grandes conquistas de la arquitectura moderna, porque permitió que una misma máquina ejecutara múltiples tareas mediante software. Sin embargo, las cargas contemporáneas han demostrado que la generalidad absoluta puede resultar ineficiente para ciertos dominios. El entrenamiento de redes neuronales profundas, el procesamiento gráfico, la criptografía, la simulación física, el análisis genómico o el procesamiento de señales pueden beneficiarse de arquitecturas especializadas que ejecutan operaciones recurrentes con mayor eficiencia energética y temporal. Esta tendencia no niega el valor del propósito general, pero evidencia que el diseño arquitectónico actual se desplaza hacia sistemas heterogéneos donde CPU, GPU, FPGA, aceleradores de inteligencia artificial y memorias especializadas colaboran según la naturaleza de la carga.

No obstante, la especialización también introduce una tensión epistemológica y técnica. Mientras una arquitectura general favorece la flexibilidad, la portabilidad y la continuidad del software, una arquitectura especializada puede incrementar el rendimiento a costa de dependencia tecnológica, complejidad de programación y menor adaptabilidad a tareas no previstas. En este punto, la arquitectura computacional contemporánea se encuentra ante un dilema: cuanto más se adapta el hardware a una carga específica, mayor puede ser su eficiencia, pero menor su universalidad. Esta tensión obliga a repensar la idea clásica de máquina general. El computador actual ya no puede concebirse como una única unidad uniforme, sino como una composición de subsistemas especializados que deben ser coordinados por capas de software, compiladores, sistemas operativos, bibliotecas y modelos de programación.

El auge de la computación en la nube puso en evidencia otra limitación del paradigma clásico: el computador dejó de ser una entidad localizada y pasó a funcionar como infraestructura distribuida. La nube permitió elasticidad, acceso bajo demanda, virtualización de recursos y escalabilidad económica, pero también desplazó el cuello de botella hacia la red, la latencia, la soberanía de datos y la dependencia de centros de datos centralizados. Shawish y Salama (2014) explican que la computación en la nube transformó la promesa de la computación como utilidad en un modelo operativo basado en recursos virtualizados, servicios accesibles por internet y provisión dinámica según demanda. Sin embargo, esa misma centralización dificulta atender aplicaciones sensibles al tiempo, al contexto y a la proximidad de los datos. Por ello, la crítica a Von Neumann no solo se refiere al interior del procesador, sino también al modo en que la arquitectura global de los sistemas distribuye memoria, procesamiento y comunicación.

Los paradigmas posnube, como edge computing, fog computing y dew computing, pueden interpretarse como respuestas a una nueva forma del cuello de botella: la distancia entre el lugar donde se generan los datos y el lugar donde se procesan. En sistemas de internet de las cosas, vehículos inteligentes, realidad aumentada, salud digital, vigilancia, robótica o ciudades inteligentes, enviar todos los datos hacia centros de datos remotos puede resultar ineficiente, costoso o inviable. Zhou, Zhang y Xiong (2017) sostienen que los paradigmas posnube

buscan mover infraestructura computacional desde centros de datos remotos hacia routers, estaciones base y servidores locales cercanos a los usuarios, con el fin de superar limitaciones de la computación centralizada y mejorar la experiencia en aplicaciones emergentes. Esta evolución muestra que la arquitectura contemporánea no solo intenta acercar datos al procesador dentro de la máquina, sino también acercar procesamiento a los datos dentro de la red.

La computación de borde, sin embargo, no elimina las dificultades; las redistribuye. Procesar cerca de la fuente reduce latencia y tráfico, pero introduce heterogeneidad de dispositivos, restricciones energéticas, problemas de seguridad, administración descentralizada y complejidad de coordinación. Li et al. (2018) advierten que los paradigmas orientados al borde surgen para enfrentar problemas de ancho de banda, sobrecarga de comunicación y falta de conciencia de ubicación en la nube, pero todavía presentan desafíos de arquitectura, gestión de recursos, disponibilidad, escalabilidad y optimización del sistema. Desde esta perspectiva, la crítica al modelo clásico debe evitar caer en una narrativa de sustitución simple. No se trata de reemplazar una arquitectura por otra, sino de reconocer que cada solución desplaza el problema hacia nuevos niveles de organización.

La misma lógica se observa en la computación de alto rendimiento. Los supercomputadores contemporáneos no pueden ser entendidos como simples versiones más grandes de un computador secuencial. Son ensamblajes complejos de nodos, redes, memorias, aceleradores, sistemas de almacenamiento, software de administración y modelos de programación paralela. Su eficiencia depende de la coordinación entre estos elementos, no de la potencia aislada de una unidad. El cuello de botella de Von Neumann se transforma allí en múltiples cuellos de botella: acceso a memoria local, comunicación entre nodos, transferencia hacia aceleradores, sincronización global, entrada/salida masiva y consumo energético. La arquitectura de alto rendimiento muestra que el problema moderno no es únicamente superar la secuencialidad, sino evitar que la complejidad de coordinación consuma las ganancias del paralelismo.

También es necesario problematizar la relación entre arquitectura computacional y producción de conocimiento. En la ciencia contemporánea, los computadores

no solo ejecutan cálculos; participan en la construcción de modelos, simulaciones, visualizaciones, predicciones y decisiones. Esto significa que las limitaciones arquitectónicas pueden influir en las formas de investigación posibles. Un modelo científico puede simplificarse porque la arquitectura disponible no permite mayor resolución; una simulación puede limitar su escala por restricciones de memoria; un algoritmo puede diseñarse para adaptarse a una GPU; una investigación puede depender de recursos cloud o HPC inaccesibles para ciertos grupos. Así, el cuello de botella no es solo técnico, sino también epistemológico: condiciona qué preguntas pueden formularse, qué modelos pueden ejecutarse y qué resultados pueden obtenerse en un tiempo razonable.

Esta dimensión epistemológica se vuelve evidente cuando se consideran paradigmas no convencionales como la computación molecular. La propuesta de emplear ADN como soporte de procesamiento no solo busca una tecnología alternativa, sino que cuestiona la identificación entre computación y arquitectura electrónica tradicional. Păun, Rozenberg y Salomaa (1998/2003) presentan la computación con ADN como una nueva forma de explotar paralelismo masivo y complementariedad molecular para abordar problemas computacionales desde un soporte distinto al silicio. Aunque estos enfoques enfrentan desafíos de precisión, control, errores e implementación práctica, su valor crítico consiste en ampliar la noción de arquitectura. Si la computación puede materializarse en diferentes medios físicos, entonces el modelo de Von Neumann debe comprenderse como una realización histórica dominante, no como el límite conceptual de lo computable.

La discusión crítica también debe reconocer que muchas respuestas arquitectónicas actuales siguen dependiendo de la herencia Von Neumann. Las CPU, GPU, sistemas en chip, servidores cloud y dispositivos de borde continúan utilizando memoria, instrucciones, datos, control y representación digital. Incluso las arquitecturas heterogéneas suelen mantener capas compatibles con modelos clásicos para preservar software, herramientas y ecosistemas. Por ello, no sería riguroso afirmar que el paradigma de Von Neumann ha sido abandonado. Más bien, se ha extendido, fragmentado, complementado y tensionado. Su vigencia se expresa en la persistencia del programa almacenado y de la abstracción instrucción-dato; sus límites se evidencian en la necesidad de

introducir memoria jerárquica, paralelismo masivo, aceleración especializada, procesamiento cercano a datos y modelos distribuidos.

En este sentido, el debate contemporáneo no debe formularse como oposición entre arquitectura clásica y arquitectura moderna, sino como análisis de continuidad crítica. La arquitectura de Von Neumann sigue siendo una base pedagógica, histórica y funcional indispensable, pero resulta insuficiente para explicar por sí sola la complejidad de los sistemas actuales. Su mayor fortaleza fue convertir la máquina en un dispositivo general y reprogramable; su mayor debilidad es haber consolidado una separación entre procesamiento y memoria que se vuelve problemática en contextos de datos masivos. Las arquitecturas actuales intentan conservar la flexibilidad del programa almacenado mientras reducen la distancia entre cómputo y datos, explotan paralelismo, especializan operaciones y distribuyen recursos a diferentes escalas.

La eficiencia, en consecuencia, ya no puede definirse como velocidad bruta. Un sistema eficiente es aquel que articula de manera equilibrada procesamiento, memoria, comunicación, energía, software y carga de trabajo. Puede ser más eficiente procesar datos en el borde que enviarlos a la nube; puede ser más eficiente usar una GPU para operaciones matriciales que una CPU general; puede ser más eficiente reorganizar datos que agregar procesadores; puede ser más eficiente rediseñar un algoritmo que adquirir hardware más potente. La arquitectura computacional contemporánea exige una racionalidad sistémica donde el rendimiento se evalúe como relación entre recursos y finalidad, no como suma de componentes poderosos.

La crítica al cuello de botella de Von Neumann permite, finalmente, comprender el sentido de la arquitectura de computadores en la era del procesamiento masivo de datos. El problema central ya no es únicamente construir máquinas capaces de ejecutar instrucciones, sino diseñar ecosistemas computacionales capaces de mover, almacenar, procesar, interpretar y devolver información con eficiencia, seguridad y sostenibilidad. Esta exigencia rebasa el modelo clásico sin anularlo. La arquitectura contemporánea se configura como un campo de negociación entre herencia y ruptura: conserva principios fundamentales de la computación

digital, pero los reordena frente a nuevas demandas de escala, velocidad, ubicuidad, inteligencia y eficiencia energética.

Desde esta perspectiva, el cuello de botella de Von Neumann debe entenderse menos como una falla puntual y más como un síntoma de una transformación histórica. La computación moderna nació bajo el ideal de una máquina general capaz de ejecutar programas almacenados; la computación actual avanza hacia sistemas distribuidos, heterogéneos, paralelos y orientados a datos. La tensión entre ambos momentos define el núcleo de la arquitectura computacional contemporánea. Diseñar sistemas eficientes implica reconocer que ningún componente aislado resuelve el problema del rendimiento. Solo la articulación entre arquitectura funcional, organización física, modelo de programación, jerarquía de memoria, interconexión, paralelismo y conocimiento del dominio permite responder a las demandas actuales de procesamiento masivo de información.

III. CONCLUSIONES

la arquitectura computacional constituye mucho más que una disposición técnica de componentes electrónicos; representa la materialización histórica de una forma de comprender el cálculo, la información y la automatización del conocimiento. A lo largo del capítulo se ha evidenciado que el computador moderno no surgió únicamente como resultado de avances en hardware, sino como consecuencia de una trayectoria intelectual y tecnológica en la que el cálculo mecánico, la lógica matemática, la teoría de la computabilidad, el programa almacenado y la organización física de la máquina convergieron para dar lugar a sistemas capaces de ejecutar procedimientos formales de manera automática. Esta perspectiva permite comprender que cada arquitectura expresa una determinada concepción del procesamiento, de la memoria, del control y del rendimiento.

El análisis de la evolución de las familias de computadores permitió reconocer que la historia de la computación no avanza mediante sustituciones absolutas, sino mediante procesos de sedimentación, recombinación y especialización. Las máquinas mecánicas exteriorizaron operaciones humanas; los computadores

electrónicos ampliaron la escala del cálculo; el paradigma de programa almacenado hizo posible la máquina de propósito general; el microprocesador democratizó el acceso al cómputo; los sistemas paralelos y de alto rendimiento respondieron a la complejidad creciente de la ciencia y la industria; y las arquitecturas distribuidas, de borde y no convencionales ampliaron el concepto mismo de computador. Esta trayectoria muestra que cada familia computacional resolvió ciertas limitaciones, pero también generó nuevas exigencias que impulsaron posteriores transformaciones.

La distinción entre arquitectura funcional y organización física resultó fundamental para comprender la doble naturaleza del computador. Por un lado, toda máquina ofrece un modelo lógico de ejecución, compuesto por instrucciones, registros, tipos de datos, modos de direccionamiento y comportamientos visibles para el software. Por otro lado, ese modelo requiere una realización material mediante circuitos, buses, memoria, señales de control, unidades funcionales y mecanismos de interconexión. Esta diferencia permite advertir que dos sistemas pueden compartir una misma arquitectura funcional y, sin embargo, alcanzar rendimientos muy distintos debido a su organización interna. En consecuencia, el diseño computacional exige articular abstracción y materialidad, porque la corrección lógica del programa solo adquiere efectividad cuando puede ejecutarse bajo condiciones físicas adecuadas.

La revisión de la taxonomía de Flynn permitió situar el paralelismo como una respuesta estructural a los límites del procesamiento secuencial. Las categorías SISD, SIMD, MISD y MIMD ofrecen un marco inicial para comprender la relación entre flujos de instrucciones y flujos de datos, aunque las arquitecturas contemporáneas exceden con frecuencia esa clasificación al combinar múltiples formas de paralelismo en sistemas heterogéneos. Esta revisión mostró que el paralelismo no debe entenderse como simple multiplicación de procesadores, sino como una estrategia de organización del trabajo computacional que exige considerar dependencias, comunicación, sincronización, memoria, granularidad y modelo de programación. Por ello, la eficiencia paralela depende menos de la cantidad de recursos disponibles que de la capacidad para coordinarlos de manera productiva.

El estudio de las métricas de rendimiento y de la Ley de Amdahl permitió establecer que el desempeño computacional es una propiedad sistémica. Tiempo de ejecución, latencia, throughput, ancho de banda, speedup, eficiencia paralela, escalabilidad, consumo energético y costo no son indicadores aislados, sino dimensiones interdependientes del diseño arquitectónico. La Ley de Amdahl recordó que toda mejora queda limitada por la fracción que no puede escalar, ya sea por secuencialidad, memoria, comunicación, sincronización o entrada y salida. Esta idea conserva plena vigencia en los sistemas actuales, porque el rendimiento real continúa condicionado por cuellos de botella que no siempre se resuelven agregando más hardware. Diseñar sistemas eficientes implica identificar la parte limitante del proceso y actuar sobre ella con criterios técnicos y algorítmicos adecuados.

La discusión crítica evidenció que el paradigma de Von Neumann mantiene una importancia histórica y conceptual indiscutible, pero también enfrenta límites frente a las demandas contemporáneas de procesamiento masivo de datos. Su mayor aporte fue permitir la máquina general reprogramable; su principal restricción radica en la separación entre procesamiento y memoria, que produce esperas, transferencias constantes y dependencia del flujo de datos. Las arquitecturas actuales no han abandonado por completo este paradigma, pero lo han complementado mediante jerarquías de memoria, paralelismo, aceleradores, sistemas distribuidos, procesamiento de borde y modelos heterogéneos. Así, la arquitectura computacional contemporánea puede entenderse como una negociación entre la herencia del programa almacenado y la necesidad de acercar el cómputo a los datos, reducir latencias y aprovechar recursos especializados.

Finalmente, el capítulo permite afirmar que los principios arquitectónicos que rigen el diseño de sistemas eficientes en la actualidad se organizan alrededor de una idea central: ninguna arquitectura puede evaluarse al margen del problema que debe resolver. La eficiencia depende de la correspondencia entre carga de trabajo, modelo funcional, organización física, jerarquía de memoria, forma de paralelismo, patrón de comunicación, consumo energético y contexto de uso. En consecuencia, el futuro de la arquitectura computacional no se orienta hacia una única máquina universalmente óptima, sino hacia sistemas flexibles, heterogéneos, escalables y sostenibles, capaces de integrar propósito general y

especialización, centralización y distribución, potencia de cálculo y proximidad a los datos. Esta comprensión proporciona la base conceptual para continuar, en los siguientes capítulos, con el análisis de las jerarquías de memoria, las interconexiones, el microprocesador moderno, la entrada y salida, la virtualización y las arquitecturas emergentes.

CAPÍTULO II: DINÁMICAS DE INTERCONEXIÓN Y JERARQUÍAS DE MEMORIA EN LA ERA DEL BIG DATA

Chapter II: Interconnection dynamics and memory hierarchies in the big data era

CAPÍTULO II: DINÁMICAS DE INTERCONEXIÓN Y JERARQUÍAS DE MEMORIA EN LA ERA DEL BIG DATA

Chapter II: Interconnection dynamics and memory hierarchies in the big data era

I. Introducción

La arquitectura y organización de computadores en la era del Big Data no puede comprenderse únicamente desde la potencia del procesador ni desde la cantidad de operaciones que una unidad central o un acelerador puede ejecutar por segundo. El rendimiento real de los sistemas contemporáneos depende, cada vez con mayor intensidad, de la capacidad para mover datos, reducir latencias, sostener ancho de banda, administrar jerarquías de memoria, garantizar disponibilidad y ubicar información en los niveles adecuados de almacenamiento. En este contexto, la comunicación interna del hardware deja de ser un aspecto secundario del diseño computacional y se convierte en una condición estructural para que el procesamiento sea efectivo. La velocidad de cálculo, por sí sola, resulta insuficiente si los datos no llegan oportunamente a los registros, a las cachés, a la memoria principal, a los aceleradores o a los nodos donde deben ser procesados. Por ello, este capítulo parte de una premisa central: en los sistemas modernos, especialmente aquellos orientados a Big Data, computación científica, inteligencia artificial y procesamiento distribuido, el verdadero límite del rendimiento suele encontrarse en la circulación de los datos y no exclusivamente en la capacidad aritmética de los procesadores.

El crecimiento exponencial del volumen de información ha modificado la forma en que se diseñan los sistemas de memoria y almacenamiento. Las aplicaciones actuales operan con flujos de datos heterogéneos, dinámicos, distribuidos y, con frecuencia, sensibles al tiempo de respuesta. Ya no se trata solamente de almacenar grandes cantidades de información, sino de garantizar que esta pueda ser recuperada, transferida, procesada, replicada y protegida bajo condiciones de baja latencia y alta disponibilidad. En el ecosistema de Big Data, la complejidad no proviene solo del tamaño de los datos, sino también de su velocidad de producción, de su variedad estructural, de las exigencias de confiabilidad y del

valor que se espera extraer de ellos. Mazumdar, Seybold, Kritikos y Verginadis (2019) explican que el Big Data presiona las tecnologías existentes porque exige soporte escalable, rápido y eficiente, al mismo tiempo que obliga a analizar patrones de acceso, comportamiento de centros de datos, latencias, ubicación de datos y costos de transferencia.

Esta transformación ha generado un desplazamiento conceptual en la arquitectura de computadores: el procesamiento ya no puede analizarse de forma aislada respecto de la memoria, la interconexión y el almacenamiento. En sistemas intensivos en datos, una operación computacional incluye una cadena de movimientos: los datos pueden residir en almacenamiento masivo, pasar a memoria principal, atravesar jerarquías de caché, llegar a registros, transferirse hacia una GPU, circular por buses como PCIe, desplazarse entre nodos de un clúster o replicarse en sistemas distribuidos para tolerancia a fallos. Cada una de estas etapas incorpora costos de latencia, ancho de banda, arbitraje, coherencia, contención y consumo energético. La consecuencia es clara: el diseño arquitectónico contemporáneo debe considerar la trayectoria completa del dato, desde su almacenamiento persistente hasta su consumo por unidades de ejecución.

En esta lógica, la latencia y el ancho de banda se convierten en dos categorías fundamentales para interpretar el rendimiento. La latencia expresa el tiempo que tarda un dato en estar disponible después de una solicitud; el ancho de banda indica la cantidad de datos que pueden transferirse en una unidad de tiempo. Aunque ambas métricas se relacionan, no son equivalentes. Un sistema puede tener alto ancho de banda y, aun así, resultar ineficiente para aplicaciones que requieren respuestas inmediatas; también puede poseer baja latencia en accesos pequeños, pero no sostener transferencias masivas. Esta diferencia resulta crucial en aplicaciones científicas y de Big Data, donde algunos procesos demandan grandes flujos continuos de información, mientras otros requieren acceso rápido a datos específicos. En consecuencia, la arquitectura debe equilibrar ambos factores mediante buses eficientes, redes de interconexión optimizadas, memorias jerárquicas, almacenamiento distribuido y estrategias de ubicación de datos.

El problema de la comunicación interna aparece con claridad en las arquitecturas aceleradas. La incorporación de GPU, FPGA, coprocesadores y aceleradores especializados ha aumentado la capacidad de cómputo disponible, pero también ha intensificado la dependencia respecto de los mecanismos de transferencia. Yellapragada (2022) señala que las aplicaciones aceleradas suelen iniciar y terminar con movimientos de datos entre CPU y GPU, y que dichos movimientos pueden convertirse en un cuello de botella incluso en programas optimizados, al punto de limitar el aprovechamiento del rendimiento máximo disponible. Esta observación revela una paradoja de la computación moderna: cuanto mayor es la capacidad de procesamiento paralelo, más crítico se vuelve el diseño de los canales que alimentan a las unidades de cómputo. Un acelerador sin datos oportunos no acelera el sistema; por el contrario, puede aumentar la complejidad sin producir beneficios proporcionales.

La problemática también se expresa en los sistemas de memoria. La jerarquía de memoria fue diseñada precisamente para reducir la distancia entre la velocidad del procesador y la lentitud relativa de los niveles inferiores de almacenamiento. Registros, cachés, memoria principal, memorias no volátiles, SSD, discos y almacenamiento archivístico conforman una estructura escalonada donde velocidad, capacidad, costo y persistencia se distribuyen de manera desigual. Zhang, Chen, Ooi, Tan y Zhang (2015) explican que la jerarquía de memoria se define por la latencia de acceso y la distancia lógica respecto de la CPU, y que los datos deben atravesar distintos niveles hasta llegar a los registros, por lo que el rendimiento de los programas intensivos en datos depende de la utilización adecuada de dicha jerarquía. Desde esta perspectiva, la memoria no es un espacio homogéneo de almacenamiento, sino una arquitectura dinámica de proximidad, reutilización y transferencia.

La expansión de los sistemas in-memory demuestra que el intento de reducir la dependencia del disco no elimina los problemas de rendimiento, sino que los desplaza hacia otros niveles. Al mantener datos en memoria principal se reduce el impacto de la entrada y salida tradicional, pero adquieren mayor relevancia la utilización de caché, los patrones de acceso, la localidad espacial y temporal, la concurrencia, la coherencia, la tolerancia a fallos y la consistencia. Zhang et al. (2015) sostienen que el crecimiento de la capacidad de memoria principal ha

impulsado el procesamiento in-memory para análisis interactivo, aunque estos sistemas son más sensibles a sobrecostos que antes quedaban ocultos en entornos dominados por disco, como la administración de memoria, el paralelismo y el control de concurrencia. Así, el paso del disco a la memoria no debe interpretarse como solución definitiva, sino como una reorganización de los cuellos de botella.

En paralelo, los sistemas de almacenamiento masivo han debido evolucionar para responder a las exigencias de disponibilidad, escalabilidad y confiabilidad. El Big Data no solo requiere guardar información, sino mantenerla accesible ante fallos, distribuirla entre nodos, replicarla estratégicamente, ubicarla cerca de los procesos que la consumen y protegerla frente a pérdidas. En este escenario, las configuraciones RAID, los sistemas de archivos distribuidos, las bases NoSQL, los almacenes en la nube y las estrategias de data placement forman parte de una misma problemática: cómo organizar grandes volúmenes de datos para que puedan ser utilizados de manera eficiente y segura. Mazumdar et al. (2019) plantean que los sistemas distribuidos de archivos ofrecen escalabilidad, tolerancia a fallos, acceso concurrente y soporte de metadatos, pero enfrentan desafíos vinculados con transparencia, confiabilidad, rendimiento, seguridad y ubicación de datos.

Por esta razón, la interconexión y la memoria deben analizarse como dimensiones complementarias. Un bus compartido puede convertirse en cuello de botella si múltiples dispositivos compiten por acceso; una red de interconexión puede degradar el rendimiento si su diámetro, ruta promedio, arbitraje o latencia no se ajustan a la carga; una caché puede mejorar el acceso si existe localidad, pero generar problemas de coherencia en multiprocesadores; una memoria DRAM puede ofrecer gran capacidad, pero requerir mecanismos de corrección de errores; un arreglo RAID puede mejorar disponibilidad, pero introducir costos de escritura, reconstrucción y administración; un sistema distribuido puede escalar, pero incrementar la complejidad de consistencia y migración de datos. La arquitectura moderna debe, por tanto, equilibrar velocidad, persistencia, proximidad, coherencia, confiabilidad y costo.

En el plano científico, estas dinámicas condicionan directamente el diseño de algoritmos. Los algoritmos contemporáneos no solo deben ser correctos desde el

punto de vista lógico, sino conscientes de la arquitectura que los ejecuta. Un algoritmo que ignora la jerarquía de memoria puede generar accesos aleatorios, fallos de caché y transferencias innecesarias; uno que no considera la red puede producir comunicación excesiva entre nodos; uno que no atiende el almacenamiento puede provocar congestión de entrada y salida; uno que no considera el movimiento CPU-GPU puede perder en transferencia lo que gana en paralelismo. Por ello, la eficiencia algorítmica ya no puede evaluarse únicamente mediante complejidad asintótica. Debe analizarse también en términos de localidad de datos, patrón de acceso, ancho de banda requerido, latencia tolerable, paralelismo disponible y costo de comunicación.

El presente capítulo examina estas tensiones a partir de tres ejes. Primero, se analizan las estructuras de buses y los protocolos de arbitraje como mecanismos fundamentales para coordinar el acceso compartido a recursos de comunicación interna. Este subtema permitirá comprender cómo los buses, las interconexiones CPU-GPU, las redes de nodos y las tecnologías ópticas o de baja latencia condicionan el rendimiento efectivo de sistemas paralelos y acelerados. Segundo, se desarrolla la jerarquía de memoria desde los registros hasta la memoria principal semiconductora, incorporando cachés, localidad, DRAM, coherencia, NUMA, confiabilidad y tecnologías emergentes. Tercero, se estudian los sistemas de almacenamiento masivo y las configuraciones RAID, relacionándolos con disponibilidad, replicación, sistemas distribuidos, ubicación de datos y gestión de grandes volúmenes de información.

La discusión crítica del capítulo contrastará la velocidad de procesamiento con la velocidad de acceso a memoria, mostrando cómo esta brecha afecta el diseño de algoritmos científicos y sistemas de Big Data. El problema central no será únicamente técnico, sino epistemológico y metodológico: la forma en que los datos se mueven, se ubican y se protegen determina qué escalas de análisis son posibles, qué simulaciones pueden ejecutarse, qué modelos pueden entrenarse y qué servicios pueden sostenerse en tiempo real. En este sentido, la arquitectura de computadores en la era del Big Data no puede ser pensada como una suma de componentes, sino como una ecología de comunicación y memoria donde cada decisión de diseño afecta la capacidad de transformar datos en conocimiento operativo.

II. DESARROLLO

Estructuras de buses y protocolos de arbitraje

Las estructuras de buses constituyen uno de los primeros niveles críticos de interconexión dentro de un sistema computacional, porque permiten coordinar el intercambio de datos, direcciones y señales de control entre procesadores, memoria, periféricos y dispositivos especializados. Aunque en una explicación elemental el bus suele presentarse como un simple canal de comunicación, en términos arquitectónicos representa una zona de negociación permanente entre demanda de transferencia, latencia, ancho de banda, arbitraje, concurrencia y contención. En los sistemas actuales, especialmente aquellos orientados a procesamiento intensivo de datos, la eficiencia del bus no depende únicamente de su frecuencia o de su ancho físico, sino también de la forma en que administra solicitudes simultáneas, prioriza accesos, organiza transacciones y evita que varios componentes degraden mutuamente su rendimiento. Cho, Choi y Cho (2006) plantean que, en plataformas System-on-Chip, la comunicación de datos en arquitecturas multiprocesador puede quedar más limitada por el ancho de banda del canal compartido que por el rendimiento interno de los procesadores, lo que convierte al bus en un posible cuello de botella del sistema.

El bus compartido tradicional responde a una lógica de simplicidad y economía: múltiples maestros y esclavos se conectan a un mismo medio físico o lógico, y un mecanismo de arbitraje decide qué agente puede utilizar el canal en cada momento. Esta organización reduce complejidad de diseño y facilita la integración de componentes, pero introduce una restricción evidente: solo una transacción puede ocupar el medio compartido en un intervalo determinado. Cuando pocos dispositivos compiten por el bus, esta limitación puede resultar aceptable; sin embargo, en sistemas con múltiples procesadores, controladores de memoria, dispositivos de entrada/salida, aceleradores, unidades DMA y periféricos de alta demanda, la contención se incrementa y el tiempo de espera puede crecer de forma significativa. En consecuencia, el bus deja de ser una infraestructura transparente y se convierte en un factor activo que condiciona el rendimiento global.

El arbitraje surge precisamente como respuesta a la necesidad de ordenar el acceso a un recurso compartido. Un protocolo de arbitraje define qué componente obtiene el control del bus, bajo qué condiciones, durante cuánto tiempo y con qué prioridad. Las políticas más simples pueden basarse en turnos fijos, prioridad estática o esquemas centralizados; otras pueden incorporar prioridades dinámicas, equidad, prevención de inanición, calidad de servicio o segmentación de transacciones. Desde una perspectiva de diseño, el arbitraje no solo debe garantizar corrección funcional, sino también rendimiento efectivo. Un árbitro demasiado simple puede favorecer latencias predecibles, pero desaprovechar ancho de banda; uno más complejo puede mejorar utilización, pero añadir sobrecarga de control. Por ello, la elección del protocolo depende del tipo de carga, del número de agentes, de la criticidad temporal de las transacciones y de la tolerancia del sistema a esperas variables.

En las arquitecturas System-on-Chip, el problema se vuelve más visible porque numerosos bloques de propiedad intelectual, procesadores, memorias y controladores conviven en un mismo chip o plataforma. La reutilización de componentes pre-verificados facilitó el diseño basado en plataformas, pero también incrementó la presión sobre los mecanismos compartidos de comunicación. Cho et al. (2006) indican que el aumento de bloques IP en un SoC puede elevar la latencia de transferencia en buses compartidos, especialmente en arquitecturas de una sola capa, y proponen modelos de latencia para estimar el ancho de banda efectivo y orientar decisiones tempranas sobre la arquitectura óptima del bus. Esta perspectiva muestra que el bus no debe analizarse únicamente como elemento físico, sino como una estructura cuyo comportamiento puede modelarse y optimizarse antes de la implementación final.

Una distinción relevante en este campo es la diferencia entre ancho de banda máximo y ancho de banda efectivo. El primero se relaciona con variables físicas como frecuencia, ancho del canal y número de líneas de transmisión; el segundo depende de factores lógicos como protocolo, tipo de transacción, latencia de arbitraje, ráfagas, esperas, direcciones, ciclos de control y comportamiento de los dispositivos conectados. En la práctica, el rendimiento observable suele estar más cerca del ancho de banda efectivo que del máximo teórico. Por ejemplo, una

transacción puede requerir ciclos adicionales para solicitar el bus, recibir concesión, transmitir dirección, transferir datos y liberar el recurso. Cuando se utilizan transferencias en ráfaga, parte de esta sobrecarga se amortiza, porque una sola concesión permite mover varios datos consecutivos. De ahí que los buses modernos hayan incorporado mecanismos de pipelining, bursts, múltiples capas o canales separados para reducir esperas y mejorar utilización.

La evolución desde buses de una sola capa hacia buses multicapa y redes en chip responde a la necesidad de aumentar concurrencia. En una arquitectura de bus único, todos los agentes comparten el mismo medio y, por tanto, cualquier transacción puede bloquear a las demás. En un bus multicapa, diferentes maestros y esclavos pueden comunicarse mediante capas o rutas parcialmente independientes, reduciendo la contención cuando las transacciones no compiten por el mismo destino. Esta transición no elimina el arbitraje, pero lo desplaza hacia una estructura más distribuida, donde pueden coexistir varios accesos simultáneos. En sistemas más complejos, las redes en chip sustituyen la idea de un único bus por una topología de routers, enlaces y protocolos de comunicación internos, acercando la arquitectura de interconexión en chip a los problemas clásicos de redes paralelas.

La relación entre bus e interconexión se vuelve todavía más crítica cuando se consideran aceleradores conectados al procesador principal. En sistemas CPU-GPU discretos, el bus PCI Express funciona como enlace entre la memoria del host y la memoria del dispositivo, de modo que el rendimiento de una aplicación acelerada depende no solo de la capacidad de la GPU, sino de la eficiencia con la que los datos atraviesan dicha interconexión. Yellapragada (2022) explica que, antes de asignar una tarea a la GPU, los datos deben copiarse desde la memoria principal de la CPU hacia la memoria global de la GPU, y que los resultados suelen regresar posteriormente al host, lo cual convierte el movimiento host-dispositivo en una dimensión crítica del tiempo total de ejecución. En este contexto, el bus opera como frontera entre dos espacios de memoria y dos dominios de procesamiento, por lo que cualquier insuficiencia en transferencia puede anular parte de la ventaja computacional del acelerador.

La problemática del PCIe no se limita a su ancho de banda nominal. El costo real de comunicación incluye latencia de inicio, sincronización, copias, preparación de buffers, alineación de datos, páginas de memoria, controladores, colas y comportamiento de la aplicación. Una GPU puede poseer cientos o miles de núcleos y una memoria interna de alto ancho de banda, pero si el programa requiere transferencias frecuentes de pequeños bloques o intercambios constantes con la CPU, el bus puede convertirse en el componente dominante del tiempo de ejecución. Wilkinson y Skylaris (2013) evidencian este problema al portar operaciones FFT de ONETEP hacia GPU, pues aunque ciertas etapas algorítmicas se aceleraron, la transferencia de datos entre la GPU y la máquina host se mantuvo como un cuello de botella significativo en la versión evaluada.

El análisis de los métodos de acceso a datos en sistemas heterogéneos muestra que la ubicación de los datos y la ruta empleada para acceder a ellos son tan importantes como el dispositivo que ejecuta el cálculo. Kalidas, Daga, Krommydas y Feng (2015) señalan que las GPU discretas suelen requerir transferencias desde memoria del host hacia memoria de GPU a través de PCIe, y que en ciertas aplicaciones este sobrecosto puede neutralizar las ganancias obtenidas por el procesamiento paralelo del dispositivo. Esta observación permite extender el concepto de arbitraje más allá del bus compartido clásico: en arquitecturas heterogéneas, también se arbitra, de manera explícita o implícita, entre rutas de acceso, espacios de memoria, políticas de copia, acceso directo, coherencia y amortización del costo de transferencia.

Las unidades de procesamiento acelerado y las arquitecturas de memoria compartida intentan reducir este problema integrando CPU y GPU en un mismo chip o permitiendo acceso directo a memoria del host. Sin embargo, estas soluciones no eliminan automáticamente los costos de acceso, porque diferentes rutas de datos pueden presentar latencias, coherencia, ancho de banda y consumos energéticos distintos. En una arquitectura APU, la unificación del espacio de direcciones simplifica programación y reduce copias explícitas, pero el rendimiento depende de la ruta concreta que sigue el dato y de si el acceso mantiene coherencia con la CPU. Kalidas et al. (2015) muestran que distintos métodos de acceso pueden afectar de manera diferenciada el rendimiento, la energía y la potencia según el patrón de comunicación y cómputo de la aplicación.

Por ello, la interconexión ya no puede pensarse como un simple canal, sino como una red de posibilidades de acceso cuyo aprovechamiento depende de la arquitectura y del comportamiento del programa.

En sistemas paralelos de mayor escala, el problema del bus se transforma en el problema de la red de interconexión. Cuando múltiples nodos de cómputo deben intercambiar mensajes o acceder a memoria remota, la topología, el diámetro, el ancho de bisección, la latencia de enrutamiento, el número de saltos, la congestión y la localidad de comunicación se vuelven determinantes. Dally (1990) sostiene que el componente crítico de un computador concurrente es su red de comunicación, porque muchos algoritmos están limitados por comunicación más que por procesamiento, especialmente cuando el grano de paralelismo disminuye y los mensajes se vuelven más frecuentes. Esta afirmación conecta directamente con la era del Big Data: a medida que los sistemas procesan más datos distribuidos, la eficiencia depende tanto de calcular como de comunicar.

La topología de interconexión define las rutas posibles entre nodos y, con ello, afecta la latencia y el throughput. Las redes k-ary n-cube, las mallas, los toros, los hipercubos, los árboles, las topologías Clos, Fat Tree, Dragonfly u otras variantes representan diferentes compromisos entre número de enlaces, diámetro, facilidad de implementación, costo físico, tolerancia a fallos y ancho de bisección. Dally (1990) analiza las redes k-ary n-cube bajo restricciones de bisección de cableado y muestra que las redes de baja dimensión pueden presentar menor latencia y mejor throughput en puntos calientes que redes de mayor dimensión cuando se mantienen constantes ciertas restricciones físicas. Esta conclusión revela que la topología no puede escogerse únicamente por elegancia matemática; debe evaluarse en función de restricciones materiales, longitud de enlaces, densidad de cableado, empaquetamiento y patrones de tráfico.

Agarwal (1991) amplía esta discusión al señalar que la latencia de una red de interconexión depende no solo de sus propiedades internas, como dimensión, ancho de canal, retraso de nodo y retraso de cable, sino también de los patrones de comunicación de las aplicaciones paralelas. Esta idea es crucial porque impide separar arquitectura y software. Una red puede ser óptima bajo tráfico uniforme, pero comportarse de manera menos eficiente cuando los accesos presentan

localidad, concentración o comunicación irregular. Del mismo modo, una topología con baja distancia promedio puede no ser suficiente si la contención se concentra en enlaces críticos. Así, el diseño de interconexiones debe considerar no solo la estructura física de la red, sino la semántica de los patrones de datos que circularán por ella.

La noción de diámetro y distancia promedio permite relacionar teoría de grafos con rendimiento arquitectónico. En una red, el diámetro expresa el mayor número mínimo de saltos entre dos nodos; la longitud promedio del camino más corto estima cuántos saltos requiere la comunicación típica. En sistemas de gran escala, reducir estas métricas puede disminuir latencia y mejorar el rendimiento de aplicaciones paralelas. Nakao, Sakai, Hanada, Murai y Sato (2021) proponen optimizar topologías de interconexión mediante teoría de grafos, buscando reducir diámetro y longitud promedio de caminos, y muestran que tales diseños pueden superar topologías aleatorias y k-ary n-cube convencionales en simulaciones de latencia, rendimiento de benchmarks paralelos y bisección. La interconexión, por tanto, no es solo infraestructura de transporte, sino objeto de optimización matemática y arquitectónica.

El diseño de redes de baja latencia también debe considerar los límites físicos de señalización. A medida que los sistemas crecen, la longitud de cables, la pérdida eléctrica, el consumo energético, la densidad de pines y el empaquetamiento imponen restricciones que afectan la elección de topología. Gupta y Dally (2005) sostienen que la selección de una topología óptima depende de las tecnologías de señalización y empaquetamiento disponibles, porque la distancia modifica el costo y el ancho de banda de los enlaces eléctricos u ópticos. Esta perspectiva demuestra que no existe una topología ideal en abstracto; la red óptima depende de un conjunto de condiciones tecnológicas, económicas y físicas que cambian con la escala del sistema.

Las interconexiones ópticas surgen como respuesta a algunas de estas restricciones. En redes eléctricas de alta velocidad, las pérdidas, la atenuación, el consumo y la necesidad de repetidores limitan el escalamiento. Las tecnologías fotónicas prometen mayor ancho de banda, menor pérdida en distancias relevantes y mejor eficiencia energética por unidad de transferencia. Shacham y

Bergman (2007) argumentan que las redes de interconexión de ultrabaja latencia se han vuelto necesarias en sistemas HPC modernos, y que los avances en integración fotónica permiten proponer arquitecturas con rutas ópticas extremo a extremo, aunque enfrentan desafíos como la ausencia de buffers ópticos eficientes y la necesidad de resolver contenciones de manera distinta. De este modo, la interconexión óptica no es simplemente un reemplazo de cables eléctricos, sino una reconfiguración del modo en que se piensa el enrutamiento, la contención y la recuperación de mensajes.

La comunicación interna, desde el bus hasta la red óptica, expresa una misma tensión: el procesamiento necesita datos, pero el movimiento de datos tiene costo. En una escala pequeña, ese costo aparece como latencia de arbitraje en un bus compartido; en una escala intermedia, como transferencia host-dispositivo en PCIe; en una escala grande, como latencia de red entre nodos; y en una escala extrema, como restricción física de cableado, energía y topología. Esta continuidad permite comprender por qué las arquitecturas modernas no pueden evaluarse solo por la potencia de sus unidades de cálculo. Un sistema computacional eficiente requiere que sus mecanismos de interconexión acompañen la demanda de datos de los procesadores, memorias, aceleradores y dispositivos de almacenamiento. Cuando esa relación se rompe, la arquitectura entra en una zona de subutilización: existen recursos de cómputo disponibles, pero el sistema no logra alimentarlos con suficiente rapidez ni coordinar su acceso de manera adecuada.

La transición desde buses compartidos hacia redes de interconexión más complejas también modifica el sentido del arbitraje. En un bus clásico, el arbitraje consiste en decidir qué agente ocupa un canal común; en una red conmutada, el problema se desplaza hacia la selección de rutas, la gestión de colas, la resolución de contención, la asignación de canales virtuales y la prevención de bloqueos. Esta evolución no elimina el principio de acceso regulado, sino que lo distribuye por la arquitectura. Un paquete o una transacción puede competir por enlaces, buffers, puertos de entrada, puertos de salida o rutas alternativas. Por ello, el arbitraje moderno ya no se limita a un árbitro central, sino que se expresa como un conjunto de decisiones locales y globales destinadas a mantener el flujo de

datos con la menor latencia posible y con el mayor aprovechamiento del ancho de banda disponible.

En las redes de alto rendimiento, la latencia no se explica por una única causa. Intervienen el tiempo de vuelo de la señal, el tiempo de serialización, el retardo de reenvío, el retardo de colas, la cantidad de saltos, el tamaño del paquete y el grado de congestión. Shacham y Bergman (2007) descomponen la latencia de mensaje en componentes asociados a propagación, serialización y enrutamiento, lo cual permite comprender que reducir la distancia física no basta si la red mantiene muchos saltos, colas o mecanismos de conmutación lentos. Esta perspectiva resulta fundamental para el diseño arquitectónico, porque obliga a analizar la comunicación como una suma de costos acumulados. Una red con enlaces rápidos puede ser ineficiente si su topología obliga a atravesar demasiados routers; una red con pocos saltos puede saturarse si no dispone de suficiente capacidad de bisección; y una red con gran ancho de banda puede degradarse si el arbitraje genera esperas prolongadas.

Las topologías ópticas basadas en enrutamiento por longitud de onda representan una alternativa relevante frente a la expansión de los sistemas de gran escala. Proietti, Cao, Nitta, Li y Yoo (2015) proponen una arquitectura óptica jerárquica para sistemas de cómputo de gran escala, apoyada en routers de rejilla de guía de onda arreglada, con el objetivo de reducir latencia, aumentar throughput y mejorar escalabilidad frente a redes electrónicas tradicionales. Lo interesante de este enfoque es que no se limita a sustituir un medio físico por otro, sino que introduce otra lógica de interconexión: la posibilidad de establecer comunicaciones all-to-all dentro de ciertos niveles jerárquicos, aprovechar el enrutamiento por longitud de onda y reducir la dependencia de conmutadores electrónicos store-and-forward. Esta clase de diseños responde a una necesidad creciente: sostener el intercambio intensivo de datos entre miles o cientos de miles de nodos sin que la red se convierta en el límite principal del sistema.

Sin embargo, la promesa de las redes ópticas debe interpretarse con cautela. Aunque ofrecen ventajas en ancho de banda, distancia y eficiencia energética, también plantean desafíos técnicos relacionados con integración, control, contención, costo de dispositivos, ausencia de buffers ópticos eficientes y

compatibilidad con infraestructuras existentes. En consecuencia, su incorporación no puede entenderse como una solución universal a los problemas de interconexión, sino como una estrategia que debe evaluarse según escala, patrón de tráfico, requerimientos de latencia, costo de implementación y madurez tecnológica. En sistemas de Big Data y HPC, donde las cargas pueden oscilar entre comunicación masiva, transferencia irregular, sincronización global o acceso remoto a datos, la red óptima será aquella que logre equilibrar capacidad, previsibilidad, energía y costo bajo condiciones concretas de uso.

La comparación entre buses internos, enlaces CPU-GPU, redes de nodos e interconexiones ópticas permite advertir que el concepto de comunicación computacional se ha expandido. En los primeros sistemas, el bus era principalmente un mecanismo de conexión entre CPU, memoria y periféricos; en las arquitecturas contemporáneas, la interconexión constituye una capa compleja de organización del rendimiento. El sistema completo puede incluir buses internos, controladores de memoria, enlaces PCIe, NVLink u otros enlaces especializados, redes en chip, redes de almacenamiento, interconexiones de clúster y redes de centro de datos. Cada una de estas capas introduce sus propias reglas de arbitraje, transferencia, contención y sincronización. La eficiencia del sistema depende de que estas capas no se comporten como compartimentos aislados, sino como una cadena coherente de circulación de datos.

En este punto, la arquitectura de interconexión se relaciona de manera directa con el diseño de algoritmos científicos. Un algoritmo que requiere comunicación global frecuente puede comportarse mal en una red con alto diámetro o bajo ancho de bisección. Un algoritmo de stencil puede beneficiarse de localidad espacial y comunicación entre vecinos, mientras que un algoritmo de aprendizaje distribuido puede requerir agregación frecuente de parámetros. Un proceso de visualización de datos masivos puede depender más de transferencia entre almacenamiento, CPU y GPU que de cómputo puro. Cao, Wu y Wang (2011) muestran que, en visualización de datos volumétricos variables en el tiempo, el cuello de botella no se encuentra únicamente en el renderizado, sino en la transferencia de datos a través de una estructura multinivel que incluye disco, memoria principal y GPU. Este tipo de evidencia confirma que la comunicación

debe incorporarse al diseño algorítmico desde el inicio y no tratarse como un problema posterior de implementación.

La arquitectura de buses y redes también se vincula con la noción de localidad. Cuando los datos se procesan cerca del lugar donde se encuentran, se reduce el tráfico global y disminuye la presión sobre los canales compartidos. Por el contrario, cuando los datos deben desplazarse repetidamente entre memoria, aceleradores, almacenamiento y nodos remotos, la interconexión se convierte en un recurso crítico. Esta relación explica por qué muchas estrategias modernas, como la compresión selectiva, el procesamiento in-memory, la computación cercana a datos, las cachés jerárquicas, la replicación inteligente o el data placement, buscan disminuir el movimiento innecesario. La reducción de transferencia no solo mejora rendimiento, sino que también reduce consumo energético y congestión.

En sistemas de almacenamiento y Big Data, la interconexión adquiere una dimensión adicional. El acceso a datos distribuidos no ocurre solo dentro de una placa o un chip, sino entre nodos, racks, centros de datos o regiones. En estos escenarios, el arbitraje puede expresarse como asignación de recursos de red, planificación de tareas, balanceo de carga, elección de réplica o ubicación de datos. Mazumdar et al. (2019) sostienen que la ubicación de datos en ecosistemas cloud-Big Data debe considerar costos, latencias, patrones de acceso, comportamiento de recursos y requisitos no funcionales de las aplicaciones. Esta perspectiva amplía el concepto de bus hacia una visión de interconexión distribuida: ya no se trata solo de quién usa una línea física, sino de dónde debe residir el dato para que su uso sea eficiente.

La siguiente tabla organiza comparativamente algunas estructuras de interconexión relevantes para la arquitectura de computadores en la era del Big Data. Su finalidad no es presentar una clasificación exhaustiva de todas las tecnologías disponibles, sino evidenciar que cada forma de comunicación responde a compromisos distintos entre latencia, ancho de banda, arbitraje, escalabilidad y limitaciones físicas.

Tabla 3

Comparación de estructuras de interconexión y factores de rendimiento

Estructura de interconexión	Nivel arquitectónico predominante	Forma de arbitraje o control	Ventaja principal	Limitación crítica	Pertinencia en Big Data y HPC
Bus compartido tradicional	Placa, SoC o sistema básico	Árbitro central, prioridad, turno o concesión de bus	Simplicidad, bajo costo y facilidad de integración	Alta contención cuando varios agentes compiten por el mismo canal	Adecuado para sistemas simples, limitado en cargas intensivas
Bus multicapa o interconexión SoC	Sistemas en chip con múltiples IP	Arbitraje por capas, rutas parcialmente independientes	Mayor concurrencia que un bus único	Complejidad de diseño y posibles contenciones por destino	Útil en SoC multimedia, embebidos y plataformas multiprocesador
PCIe y enlaces CPU-GPU discretos	Comunicación host-dispositivo	Controladores, colas, transferencias explícitas o acceso directo	Conecta aceleradores de alto rendimiento con CPU y memoria host	Transferencias host-dispositivo pueden anular ganancias de cómputo	Crítico en IA, simulación, visualización y aceleración científica
Redes k-ary n-cube, mallas y toros	Multiprocesadores y supercomputadores	Enrutamiento, buffers, canales, control de congestión	Regularidad topológica y facilidad relativa de implementación	Diámetro, número de saltos y restricciones físicas de cableado	Relevantes en arquitecturas paralelas y sistemas de comunicación local
Topologías optimizadas por grafos	Clústeres y sistemas HPC de gran escala	Selección de rutas y optimización de diámetro y ASPL	Reducción de distancia promedio y posible mejora de latencia	Complejidad de diseño, cableado y empaquetamiento	Útiles para mejorar rendimiento de benchmarks paralelos y redes masivas
Redes ópticas y fotónicas	HPC, centros de datos y sistemas futuros	Longitud de onda, conmutación óptica, control de contención	Alto ancho de banda, baja pérdida y potencial eficiencia energética	Falta de buffers ópticos maduros, costo e integración tecnológica	Prometedoras para sistemas de muy alta escala y baja latencia
Interconexiones distribuidas cloud-Big Data	Centros de datos, nubes y almacenamiento distribuido	Planificación de tareas, selección de réplica, data placement	Escalabilidad, disponibilidad y distribución geográfica	Latencia, consistencia, tráfico intercentro y costo de transferencia	Fundamental para almacenamiento, analítica y servicios de datos masivos

Nota. *Elaboración propia a partir del análisis de Cho, Choi y Cho (2006), Dally (1990), Agarwal (1991), Nakao et al. (2021), Gupta y Dally (2005), Shacham y*

Bergman (2007), Proietti et al. (2015), Yellapragada (2022), Kalidas et al. (2015), Cao et al. (2011) y Mazumdar et al. (2019).

La tabla permite observar que no existe una estructura de interconexión superior en todos los escenarios. Un bus compartido puede ser eficiente por costo y simplicidad en sistemas pequeños, pero inadecuado en plataformas con múltiples agentes de alta demanda. Una red k-ary n-cube puede ofrecer regularidad y control de implementación, pero su rendimiento depende de dimensión, bisección, longitud de enlaces y tráfico. Las topologías optimizadas por grafos pueden reducir diámetro y distancia promedio, aunque exigen resolver problemas de cableado y empaquetamiento. Las interconexiones ópticas prometen saltos importantes en ancho de banda y energía, pero arrastran desafíos de integración y control. Los sistemas cloud-Big Data, por su parte, trasladan la interconexión al plano de la ubicación y movimiento de datos entre infraestructuras distribuidas.

Este análisis muestra que los buses y protocolos de arbitraje deben entenderse como parte de un continuo arquitectónico. En un extremo se encuentra el bus compartido, donde la disputa se concentra en un medio común; en otro, las redes distribuidas de gran escala, donde la disputa se expresa en rutas, nodos, enlaces, réplicas y ubicaciones de datos. Entre ambos aparecen PCIe, redes en chip, interconexiones de aceleradores, topologías HPC y redes ópticas. En todos los casos, el problema de fondo es el mismo: cómo permitir que múltiples agentes accedan a datos y recursos de comunicación sin que la contención destruya el rendimiento.

Por ello, el diseño de interconexiones en la era del Big Data debe orientarse por una lógica de adecuación. No basta con seleccionar el enlace más rápido ni la topología con menor diámetro teórico. Es necesario analizar la carga, el volumen de datos, la granularidad del procesamiento, la frecuencia de comunicación, la localidad, la tolerancia a latencia, el costo energético, la escalabilidad esperada y el grado de heterogeneidad del sistema. Las arquitecturas modernas requieren que el bus, la red, la memoria y el almacenamiento sean diseñados como una unidad funcional. Cuando esta integración no ocurre, las unidades de

procesamiento quedan subordinadas a esperas, congestiones y transferencias improductivas.

En consecuencia, las estructuras de buses y los protocolos de arbitraje revelan uno de los principios centrales de la arquitectura computacional contemporánea: el rendimiento no se genera únicamente en la unidad de cálculo, sino en la coordinación eficiente de los caminos que permiten que los datos circulen hacia ella y desde ella. En la era del Big Data, la interconexión se convierte en una infraestructura epistemológica del procesamiento, porque condiciona la escala de los problemas que pueden resolverse, el tiempo en que pueden analizarse los datos y la disponibilidad real de la información para la toma de decisiones científicas, empresariales o sociales.

Jerarquía de memoria: de los registros a la memoria principal semiconductora

La jerarquía de memoria constituye una de las respuestas arquitectónicas más relevantes frente a la brecha histórica entre la velocidad de procesamiento y la velocidad de acceso a los datos. A medida que los procesadores incrementaron su capacidad de ejecución, la memoria principal no avanzó al mismo ritmo en términos de latencia, lo que produjo una distancia creciente entre la rapidez con que la CPU puede operar y la rapidez con que el sistema puede suministrarle datos e instrucciones. Esta brecha no se resuelve mediante un único tipo de memoria, porque ningún dispositivo ofrece simultáneamente máxima velocidad, gran capacidad, bajo costo, persistencia y bajo consumo energético. La jerarquía surge, precisamente, como una organización escalonada que combina niveles de almacenamiento con propiedades distintas, ubicando cerca del procesador los recursos más rápidos y costosos, y alejando progresivamente aquellos más lentos, amplios y económicos.

En su nivel más inmediato, los registros representan el espacio de almacenamiento directamente manipulable por la CPU. Son pequeños, extremadamente rápidos y estrechamente vinculados con la ejecución de instrucciones. Desde el punto de vista funcional, los registros contienen operandos, direcciones, resultados temporales, contadores, estados y valores

intermedios necesarios para que la unidad aritmético-lógica y las unidades de control ejecuten operaciones. Sin embargo, su capacidad es muy reducida, por lo que no pueden alojar grandes estructuras de datos. Su importancia radica en que ningún dato puede ser procesado de manera efectiva si no llega, directa o indirectamente, a este nivel. Zhang, Chen, Ooi, Tan y Zhang (2015) explican que en las arquitecturas modernas los datos deben pasar por distintas capas de memoria hasta alcanzar los registros, ya que las instrucciones de máquina manipulan directamente la información en ese nivel inmediato de almacenamiento.

Debajo de los registros se ubica la memoria caché, concebida como un puente entre la velocidad de la CPU y la mayor latencia de la memoria principal. La caché almacena copias de datos e instrucciones recientemente utilizados o previsiblemente necesarios, con el propósito de reducir accesos a niveles inferiores. Su eficacia depende de dos principios fundamentales: localidad temporal y localidad espacial. La localidad temporal supone que un dato usado recientemente tiene alta probabilidad de volver a utilizarse en el corto plazo; la localidad espacial indica que los datos cercanos a una dirección accedida también pueden ser requeridos próximamente. Estos principios permiten que la caché no sea simplemente una memoria rápida, sino un mecanismo predictivo de proximidad, capaz de anticipar patrones de acceso y amortiguar la diferencia entre procesador y DRAM.

La estructura multinivel de la caché responde a la imposibilidad de construir una memoria única que sea simultáneamente rápida y amplia. La caché L1 suele ser la más cercana y veloz, pero también la más pequeña; la L2 ofrece mayor capacidad con algo más de latencia; la L3, cuando existe, suele funcionar como una caché de último nivel compartida o parcialmente compartida entre núcleos. Esta organización reproduce, en escala reducida, la lógica general de la jerarquía: cuanto más cerca del procesador se encuentra un nivel, menor es su latencia y capacidad, pero mayor su costo por bit. Alkhamisi (2022) describe la caché como una memoria de alta velocidad ubicada entre la CPU y la memoria principal, organizada en niveles L1, L2 y L3, cuya función es reducir el tiempo de acceso mediante el almacenamiento de datos usados con frecuencia.

La memoria caché opera mediante unidades de transferencia denominadas líneas de caché. Cuando el procesador solicita un dato, no necesariamente se trae un único byte o palabra, sino un bloque contiguo que puede contener datos cercanos. Esta decisión aprovecha la localidad espacial, pero también introduce riesgos: si el patrón de acceso es irregular, la caché puede llenarse de datos que no serán utilizados. A su vez, cuando un dato solicitado se encuentra en caché se produce un acierto; cuando no está disponible se genera un fallo, obligando a buscarlo en un nivel inferior con mayor latencia. En aplicaciones intensivas en datos, la tasa de fallos de caché puede afectar drásticamente el rendimiento, porque el procesador puede permanecer esperando datos aun cuando sus unidades de ejecución estén disponibles.

El diseño de caché también implica decisiones sobre mapeo, reemplazo y escritura. Una caché de mapeo directo asigna cada bloque de memoria a una única ubicación posible, lo que simplifica el acceso, pero puede producir conflictos frecuentes. Una caché asociativa permite mayor flexibilidad, aunque incrementa complejidad de búsqueda. Las políticas de reemplazo deciden qué línea debe expulsarse cuando la caché está llena, y las políticas de escritura determinan si los cambios se actualizan inmediatamente en niveles inferiores o se difieren. Estas decisiones revelan que la caché no es solo un dispositivo de almacenamiento, sino una arquitectura de mediación entre velocidad, coherencia, costo y predicción de comportamiento.

En sistemas multiprocesador y multinúcleo, la caché introduce un problema adicional: la coherencia. Cuando varios núcleos conservan copias de un mismo dato en sus cachés privadas o parcialmente compartidas, una modificación realizada por uno de ellos puede dejar obsoletas las copias de los demás. Si el sistema no garantiza mecanismos de coherencia, distintos procesadores podrían observar versiones diferentes de la misma información, comprometiendo la corrección del programa. Alkhamisi (2022) señala que el problema de coherencia de caché aparece cuando múltiples procesadores comparten una memoria común y mantienen copias de instrucciones u operandos en sus cachés, de modo que una actualización no reflejada en las demás puede producir disparidad de datos.

La coherencia de caché demuestra que la jerarquía de memoria no solo busca mejorar rendimiento, sino preservar una ilusión funcional de consistencia. Desde el punto de vista del programador, la memoria compartida debería comportarse como un espacio lógico coherente; desde el punto de vista físico, los datos pueden estar replicados en múltiples niveles, núcleos y dispositivos. Para mantener esa coherencia se emplean protocolos que invalidan, actualizan o coordinan estados de las líneas de caché. Estos mecanismos tienen un costo, porque generan tráfico, mensajes de control y posibles esperas. Así, una estrategia diseñada para reducir latencia puede producir nuevas formas de comunicación interna. La arquitectura debe equilibrar el beneficio de tener datos cerca del procesador con el costo de mantenerlos consistentes.

La memoria principal, generalmente basada en DRAM, ocupa un nivel intermedio entre la rapidez de la caché y la persistencia del almacenamiento secundario. Su función es alojar datos e instrucciones en ejecución, ofrecer un espacio direccionable amplio y permitir que el sistema operativo, los procesos y las aplicaciones trabajen con volúmenes mayores que los disponibles en caché. A diferencia de los registros y la caché, la memoria principal posee mayor capacidad, pero también mayor latencia. Además, su carácter volátil exige que los datos importantes sean respaldados en almacenamiento persistente o protegidos mediante mecanismos de tolerancia a fallos. Esta condición adquiere particular relevancia en sistemas in-memory, donde grandes porciones de la base de datos o de las estructuras analíticas residen en DRAM para evitar el costo de acceso a disco.

La centralidad de la DRAM en la era del Big Data se explica por la necesidad de reducir la dependencia de almacenamiento mecánico o secundario. Mantener datos en memoria permite acelerar consultas, análisis interactivos, procesamiento en tiempo real y operaciones de baja latencia. No obstante, esta ventaja no suprime otras fuentes de sobrecosto. Los sistemas in-memory deben atender problemas de indexación, distribución, concurrencia, fallos, recuperación, consistencia, localidad de datos y presión sobre la caché. En otras palabras, al eliminar o reducir el cuello de botella del disco, el diseño arquitectónico desplaza el foco hacia la eficiencia con que se usa la memoria principal y sus niveles superiores.

La relación entre memoria principal y almacenamiento persistente se vuelve especialmente compleja cuando los datos superan la capacidad disponible en DRAM. En esos casos, el sistema debe decidir qué mantener en memoria, qué mover a almacenamiento secundario, qué comprimir, qué replicar y qué recuperar bajo demanda. Esta gestión no puede resolverse mediante una política fija, porque los patrones de acceso cambian con la aplicación, la carga, el tiempo y la distribución de usuarios. La jerarquía de memoria se transforma así en una arquitectura dinámica de selección: no todos los datos merecen estar cerca del procesador, pero los datos críticos deben situarse donde su acceso no afecte el tiempo de respuesta esperado.

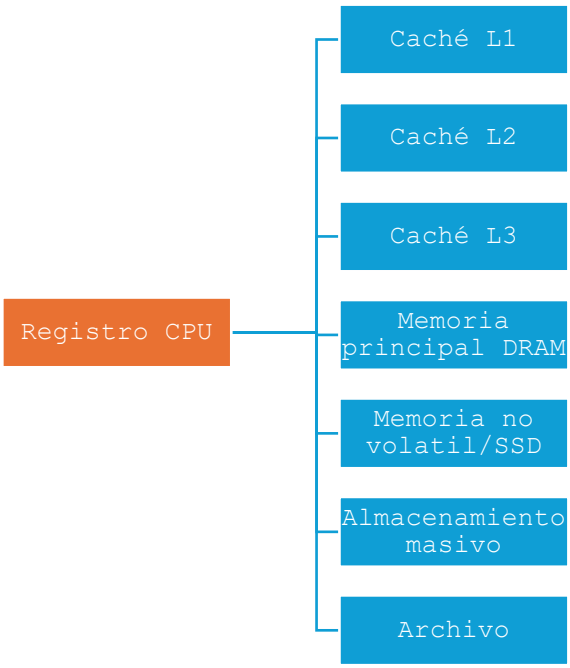
Las memorias no volátiles emergentes, como PCM, ReRAM, MRAM y otras formas de NVM, buscan ocupar espacios intermedios entre DRAM y almacenamiento persistente tradicional. Su atractivo proviene de la posibilidad de combinar mayor persistencia que DRAM con menor latencia que discos o ciertos SSD. No obstante, estas tecnologías presentan desafíos de resistencia, costo, integración, programación, durabilidad y modelos de consistencia. Vaithianathan (2025) plantea que las tecnologías emergentes de memoria ofrecen alternativas no volátiles capaces de reducir brechas de velocidad y capacidad entre DRAM y NAND flash, pero su integración en HPC requiere estrategias de optimización de jerarquía, gestión dinámica y adaptación a patrones de acceso.

En entornos de alto rendimiento, la jerarquía de memoria debe analizarse junto con la organización de múltiples núcleos, múltiples sockets y acceso no uniforme a memoria. En arquitecturas NUMA, no todos los núcleos acceden a todas las regiones de memoria con la misma latencia. La memoria local de un socket puede ser más rápida para los núcleos asociados, mientras que la memoria remota requiere atravesar interconexiones internas. Esta característica obliga a que el sistema operativo, los compiladores, los entornos de ejecución y los algoritmos consideren la ubicación de los datos. Un programa que ignora NUMA puede sufrir degradación de rendimiento por accesos remotos frecuentes, aun cuando la memoria total disponible sea amplia.

La siguiente figura sintetiza la lógica general de la jerarquía de memoria. Su propósito es representar visualmente la relación inversa entre proximidad al procesador y capacidad, así como el gradiente de latencia, costo y persistencia que estructura las decisiones arquitectónicas. No debe entenderse como una descripción única de todas las plataformas, sino como un esquema conceptual aplicable a sistemas contemporáneos que combinan registros, cachés, DRAM, memorias no volátiles y almacenamiento persistente.

Figura 2

Jerarquía de memoria y gradiente de latencia, capacidad y costo



Nota. Elaboración propia a partir de Zhang et al. (2015), Alkhamisi (2022), Vaithianathan (2025) y Kozar et al. (2025).

La figura permite observar que la jerarquía de memoria se organiza mediante compromisos inevitables. Los niveles superiores ofrecen rapidez, pero no pueden contener grandes volúmenes de datos; los niveles inferiores amplían capacidad y persistencia, pero incrementan latencia. Esta tensión explica por qué los sistemas dependen de políticas de caché, precarga, reemplazo, paginación, compresión, replicación y ubicación de datos. También permite comprender por qué el Big Data tensiona la arquitectura: los conjuntos masivos de información tienden a residir en niveles inferiores o distribuidos, mientras que el procesamiento

eficiente exige traer partes relevantes hacia niveles superiores sin saturar buses, redes ni memoria principal.

La jerarquía de memoria no solo organiza dispositivos, sino temporalidades de acceso. Los registros operan en la escala inmediata de la instrucción; la caché anticipa reutilización próxima; la DRAM sostiene el trabajo activo del proceso; la memoria no volátil busca persistencia con menor latencia que el almacenamiento tradicional; y los sistemas masivos preservan grandes volúmenes para análisis posterior, respaldo o recuperación. En la era del Big Data, la eficiencia consiste en lograr que el dato se ubique en el nivel adecuado en el momento adecuado. Un dato crítico demasiado lejos del procesador introduce espera; un dato irrelevante demasiado cerca consume recursos valiosos. La arquitectura de memoria, por tanto, se convierte en una política de proximidad selectiva.

La jerarquía de memoria también debe analizarse desde la confiabilidad. En los sistemas actuales, la memoria principal no solo debe ser rápida y amplia, sino también capaz de preservar la integridad de los datos frente a errores físicos, interferencias, degradación tecnológica, variaciones térmicas y fallos de dispositivos. Esta dimensión adquiere especial relevancia en centros de datos, HPC, inteligencia artificial y Big Data, donde un error silencioso puede afectar resultados científicos, decisiones automatizadas o procesos críticos. La reducción progresiva del tamaño de las celdas DRAM ha permitido incrementar capacidad, pero también ha intensificado preocupaciones sobre confiabilidad, energía y comportamiento físico de los dispositivos. Patel, Kim, Hassan y Mutlu (2019) explican que la DRAM sigue siendo un componente esencial por su bajo costo por bit frente a otras tecnologías, aunque las generaciones recientes han incrementado capacidad sin lograr mejoras equivalentes en rendimiento, eficiencia energética y confiabilidad, convirtiéndola en un cuello de botella importante de los sistemas modernos.

La corrección de errores en memoria se ha vuelto una estrategia indispensable para sostener sistemas confiables. Los códigos ECC permiten detectar y corregir ciertos errores antes de que se propaguen hacia el software o los resultados de la aplicación. Sin embargo, la incorporación de ECC dentro del propio chip de memoria introduce una nueva complejidad: ciertos errores pueden quedar

ocultos al sistema, y el comportamiento real de la DRAM puede ser difícil de caracterizar desde fuera. Patel et al. (2019) muestran que el ECC incorporado en dispositivos DRAM modernos puede obfuscaar las distribuciones reales de errores, porque el sistema observa únicamente errores posteriores a la corrección y no los errores originales producidos por los mecanismos físicos internos. Este hallazgo tiene implicaciones profundas para el diseño de memoria, ya que la confiabilidad no depende solo de corregir errores, sino también de comprenderlos, modelarlos y anticipar su comportamiento bajo distintas condiciones de operación.

La confiabilidad de memoria también se relaciona con el diseño de módulos y con los costos de protección. En servidores y plataformas de nube, la protección ECC debe equilibrar capacidad, costo, consumo, rendimiento y tolerancia a fallos. Criss, Bains, Agarwal, Bennett, Grunzke, Kim, Chung y Jang (2020) explican que DDR5 incorpora características orientadas a mejorar la confiabilidad mediante la limitación o acotamiento de ciertos patrones de falla, con el propósito de reducir la sobrecarga requerida por esquemas ECC capaces de corregir fallos mayores. Esta propuesta evidencia que la jerarquía de memoria no solo se optimiza por velocidad, sino también por integridad. Un sistema de alto rendimiento que no preserva datos confiables puede producir resultados rápidos pero inválidos, lo cual resulta inaceptable en aplicaciones científicas, financieras, biomédicas o gubernamentales.

La confiabilidad no se limita a la corrección de errores en DRAM. En arquitecturas multicapa, cada nivel de memoria y almacenamiento introduce sus propios riesgos: pérdida de datos en memorias volátiles, desgaste en memorias no volátiles, fallos de dispositivos, errores de transferencia, corrupción silenciosa, inconsistencias entre réplicas y fallos correlacionados en centros de datos. Por ello, la jerarquía moderna debe combinar mecanismos locales y globales de protección. A nivel de caché, se requieren políticas de coherencia; a nivel de memoria principal, códigos de corrección y refresco; a nivel de almacenamiento, redundancia, replicación o RAID; a nivel distribuido, tolerancia a fallos, checkpoints, logs y recuperación. La arquitectura de memoria se convierte así en una estructura de confianza graduada, donde cada nivel debe contribuir tanto al rendimiento como a la preservación de la información.

El problema de la jerarquía se vuelve más exigente en aplicaciones de Big Data porque el tamaño de los datos supera con frecuencia la capacidad de los niveles rápidos. Cuando una aplicación trabaja con datos que no caben completamente en caché o en memoria principal, el rendimiento depende de políticas de partición, compresión, paginación, almacenamiento temporal, acceso secuencial, prefetching y procesamiento por bloques. En estos casos, la eficiencia no proviene únicamente de tener más memoria, sino de organizar el acceso de manera que el sistema reutilice datos próximos, reduzca movimientos redundantes y mantenga en niveles superiores aquello que será utilizado con mayor frecuencia. La memoria deja de ser un depósito pasivo y se transforma en una estrategia activa de administración del flujo de datos.

El procesamiento in-memory ilustra bien esta transformación. Al ubicar los datos en DRAM, los sistemas buscan eliminar o reducir el costo del acceso a disco, especialmente en consultas interactivas, análisis en tiempo real y aplicaciones que exigen respuestas en milisegundos. No obstante, la gestión in-memory no significa que todos los problemas desaparezcan. Cuando los datos residen en memoria, las nuevas fuentes de ineficiencia se trasladan hacia caché, concurrencia, indexación, partición, control transaccional, recuperación y consistencia. Zhang et al. (2015) sostienen que los sistemas in-memory deben optimizar índices, diseños de datos, paralelismo, control de concurrencia, procesamiento de consultas, tolerancia a fallos y manejo de desbordamiento de datos, precisamente porque al reducir el cuello de botella del disco emergen otros costos antes menos visibles.

La organización física de los datos dentro de la memoria tiene un impacto directo en el rendimiento. Diseños por filas, columnas o estructuras híbridas responden a necesidades distintas. Un diseño por filas puede ser conveniente para operaciones transaccionales que acceden a registros completos; un diseño columnar puede favorecer análisis que procesan atributos específicos en grandes volúmenes; un diseño híbrido puede intentar equilibrar ambas necesidades. Esta diferencia muestra que la jerarquía de memoria no solo depende del hardware, sino también de la representación de los datos. Un mismo hardware puede ofrecer rendimientos muy distintos según la forma en que los datos se organicen, compriman, alineen, indexen y recorran. Por ello, la arquitectura de memoria y

el diseño de bases de datos deben entenderse como dimensiones interdependientes del rendimiento.

En sistemas heterogéneos, la jerarquía de memoria adquiere una forma aún más compleja porque CPU, GPU y aceleradores pueden tener memorias distintas, anchos de banda diferentes y mecanismos específicos de acceso. Una GPU suele disponer de memoria global con alto ancho de banda, pero acceder a datos del host puede ser costoso si se realiza por PCIe u otro enlace limitado. Kozar et al. (2025) explican que el procesamiento de consultas en hardware heterogéneo enfrenta retos relacionados con distribución de carga y transferencia de datos, y que tecnologías como NVLink buscan mejorar la eficiencia del intercambio entre CPU y GPU para procesar conjuntos de datos más grandes. Esto indica que la jerarquía moderna ya no puede representarse únicamente como una pirámide dentro de la CPU, sino como un conjunto de jerarquías interconectadas que deben coordinarse.

La memoria de los aceleradores plantea una tensión entre especialización y movimiento de datos. Si una operación se adapta bien a GPU, el cálculo puede acelerarse considerablemente; pero si la transferencia hacia la GPU consume demasiado tiempo, el beneficio se reduce o desaparece. Esta relación obliga a diseñar algoritmos que minimicen movimientos, agrupen operaciones, reutilicen datos en el dispositivo y eviten transferencias pequeñas y frecuentes. La jerarquía de memoria, por tanto, se convierte en un criterio algorítmico: no basta con decidir qué operación ejecutar, sino dónde ejecutarla, dónde ubicar los datos antes de ejecutarla y cuánto cuesta moverlos después. En el contexto del Big Data, esta decisión puede determinar si un sistema logra análisis interactivo o queda atrapado en esperas de transferencia.

La optimización de jerarquía también implica estrategias de prefetching y reconfiguración dinámica. El prefetching intenta anticipar datos que serán requeridos próximamente, mientras que la reconfiguración busca adaptar cachés, TLB u otros recursos según el comportamiento de la carga. Vaithianathan (2025) destaca estrategias como la reconfiguración dinámica de caché, la integración de tecnologías emergentes y las jerarquías sensibles al comportamiento de la aplicación, orientadas a reducir penalizaciones por fallos, mejorar throughput y

disminuir consumo energético en HPC. Estas estrategias tienen una lectura más amplia: la memoria eficiente no es necesariamente la memoria más grande, sino aquella que se adapta al patrón de acceso y administra mejor la distancia entre dato y procesador.

La localidad se mantiene como principio articulador de toda la jerarquía. Un sistema eficiente intenta que los datos más usados permanezcan cerca de las unidades de ejecución, que los datos relacionados se ubiquen juntos y que las transferencias aprovechen bloques completos. Los algoritmos científicos, por tanto, deben diseñarse para respetar la localidad tanto como sea posible. Operaciones matriciales, simulaciones de mallas, consultas analíticas, aprendizaje automático y visualización pueden beneficiarse cuando los datos se recorren de manera secuencial, particionada o bloqueada. Por el contrario, accesos aleatorios, estructuras dispersas, punteros encadenados o comunicación irregular pueden degradar el rendimiento por fallos de caché y accesos remotos. La jerarquía de memoria no castiga la complejidad matemática, sino la desorganización del acceso.

En la era del Big Data, la memoria principal semiconductora ocupa un lugar paradójico. Por un lado, se ha convertido en una condición para análisis de baja latencia, procesamiento in-memory y sistemas interactivos. Por otro, sigue siendo insuficiente frente al crecimiento de los datos, vulnerable por su volatilidad y limitada por la brecha con el procesador. Esta situación explica la aparición de arquitecturas híbridas que combinan DRAM, NVM, SSD y almacenamiento distribuido, intentando ubicar cada dato según su temperatura, frecuencia de uso, criticidad y costo de recuperación. La memoria caliente puede permanecer cerca del procesador; la memoria tibia puede situarse en niveles intermedios; la información fría puede desplazarse hacia almacenamiento masivo o archivístico. Este modelo de estratificación de datos es central para sistemas de alta disponibilidad.

La jerarquía de memoria también condiciona la sostenibilidad energética. Mover datos entre niveles consume energía; acceder a memoria remota consume más que operar sobre datos ya disponibles; mantener grandes volúmenes en DRAM exige alimentación constante; y sostener centros de datos con memoria

abundante requiere refrigeración, energía y administración. Por ello, optimizar memoria no significa solamente acelerar programas, sino reducir transferencias, evitar duplicaciones innecesarias, explotar localidad, comprimir cuando sea pertinente y usar tecnologías adecuadas para cada nivel. En HPC y Big Data, la eficiencia energética se vuelve parte de la eficiencia computacional.

La gestión de memoria en sistemas de alta disponibilidad exige, además, articular rendimiento con recuperación. Si los datos residen en DRAM para acelerar consultas, deben existir mecanismos de snapshot, logging, replicación o respaldo que permitan restaurarlos ante fallos. Si se utilizan memorias no volátiles, deben considerarse sus propiedades de persistencia, desgaste y consistencia. Si se distribuyen datos entre nodos, es necesario mantener réplicas y controlar fallos parciales. Esta dimensión conecta directamente con el siguiente subtema del capítulo: los sistemas de almacenamiento masivo y las configuraciones RAID. La jerarquía de memoria termina donde comienza el almacenamiento persistente, pero en los sistemas actuales esa frontera se ha vuelto difusa.

En suma, la jerarquía de memoria es una arquitectura de compromisos entre velocidad, capacidad, costo, energía, persistencia y confiabilidad. Desde los registros hasta la memoria principal semiconductora, cada nivel existe porque los demás no pueden satisfacer todas las exigencias del sistema. Los registros permiten ejecución inmediata; las cachés reducen latencia mediante localidad; la DRAM sostiene el trabajo activo; las memorias emergentes intentan cubrir la brecha entre volatilidad y persistencia; y los niveles inferiores preservan grandes volúmenes. En la era del Big Data, esta jerarquía debe pensarse como una infraestructura estratégica: su diseño determina no solo el rendimiento de los procesadores, sino la posibilidad de convertir datos masivos en información procesable dentro de tiempos útiles.

Sistemas de almacenamiento masivo y configuraciones RAID

Los sistemas de almacenamiento masivo constituyen el nivel persistente de la arquitectura computacional, allí donde los datos dejan de ser únicamente objetos de procesamiento activo y se convierten en activos que deben conservarse, recuperarse, protegerse, replicarse y administrarse durante ciclos prolongados.

En la era del Big Data, este nivel adquiere una importancia estratégica porque los volúmenes de información superan con frecuencia la capacidad de la memoria principal, de las cachés e incluso de los dispositivos locales de una sola máquina. El almacenamiento ya no puede entenderse como una extensión pasiva de la memoria, sino como una infraestructura compleja que articula rendimiento, disponibilidad, tolerancia a fallos, escalabilidad, costo y gobernanza de datos. Qin (2025) sostiene que el crecimiento de datos estructurados, semiestructurados y no estructurados ha generado cuellos de botella en los modelos tradicionales de almacenamiento de una sola máquina, especialmente cuando se trabaja con escalas de petabytes, alta concurrencia y exigencias de procesamiento en tiempo real.

La diferencia entre memoria y almacenamiento no se reduce a la velocidad de acceso. La memoria principal sostiene el trabajo activo del sistema, pero su volatilidad y costo impiden que sea el único repositorio de datos. El almacenamiento masivo, en cambio, privilegia capacidad, persistencia y recuperación, aunque a costa de mayor latencia. Esta diferencia obliga a construir mecanismos de transferencia entre niveles, políticas de caché, estrategias de respaldo, replicación y recuperación ante fallos. En aplicaciones científicas y de visualización, esta relación resulta evidente: los datos pueden residir en disco, trasladarse a memoria principal, transferirse hacia una GPU y finalmente renderizarse. Cao, Wu y Wang (2011) muestran que, en visualización volumétrica de datos variables en el tiempo, el cuello de botella principal puede desplazarse hacia la carga y transferencia de datos desde almacenamiento masivo hacia CPU y GPU, debido a la diferencia de capacidad y ancho de banda entre disco, memoria principal y memoria gráfica.

El almacenamiento masivo tradicional se apoyó durante décadas en discos magnéticos, cuya principal fortaleza fue la capacidad a bajo costo. Sin embargo, su latencia mecánica y sus límites de acceso aleatorio los volvieron insuficientes para muchas aplicaciones modernas. Las unidades de estado sólido redujeron parte de esta brecha al eliminar componentes mecánicos, mejorar operaciones de entrada/salida y aumentar tasas de transferencia, pero no resolvieron por completo el problema de la distancia entre almacenamiento persistente y procesamiento. En Big Data, el desafío no consiste solo en leer o escribir más

rápido, sino en administrar conjuntos distribuidos, soportar múltiples usuarios, garantizar disponibilidad y permitir que los datos sean procesados cerca de donde residen. Por ello, el almacenamiento masivo contemporáneo combina tecnologías físicas, sistemas de archivos, bases de datos distribuidas, políticas de replicación y mecanismos de ubicación inteligente.

Las configuraciones RAID representan una respuesta clásica al problema de combinar varios dispositivos físicos para mejorar rendimiento, capacidad o confiabilidad. El principio general consiste en organizar múltiples discos como una unidad lógica, distribuyendo o replicando datos según el nivel RAID empleado. RAID 0, por ejemplo, utiliza striping para distribuir bloques entre discos y mejorar throughput, pero no ofrece redundancia. RAID 1 replica datos para mejorar disponibilidad, aunque duplica el costo de almacenamiento. RAID 5 y RAID 6 incorporan paridad distribuida para tolerar fallos con menor sobrecarga que la duplicación completa, aunque introducen costos de cálculo y escritura. RAID 10 combina espejado y striping para equilibrar rendimiento y tolerancia a fallos. La lógica general de RAID muestra que el almacenamiento no se optimiza en una sola dimensión; toda mejora en velocidad, disponibilidad o capacidad implica un compromiso.

En entornos de Big Data, RAID conserva relevancia conceptual y operativa, pero ya no basta como único mecanismo de protección. Los sistemas distribuidos amplían la redundancia más allá de un arreglo local de discos y la trasladan hacia nodos, racks, clústeres o centros de datos. No obstante, RAID sigue siendo importante porque actúa en el nivel físico o local de protección, especialmente en servidores y sistemas de almacenamiento que deben reducir la probabilidad de pérdida por fallos de disco. Xie (2008) analizó estrategias de ubicación de datos en arreglos RAID con sensibilidad energética, lo que evidencia que incluso las técnicas clásicas de striping y redundancia deben repensarse a la luz de consumo, transferencia y organización de datos en sistemas modernos.

La configuración RAID también permite comprender una tensión más amplia: la disponibilidad puede obtenerse mediante redundancia, pero la redundancia consume capacidad, energía y tiempo de reconstrucción. En RAID 1, la recuperación es sencilla porque existe una copia directa; en RAID con paridad, la

recuperación exige recalcular datos a partir de bloques restantes, lo que puede afectar rendimiento durante la reconstrucción. En arreglos de gran capacidad, el tiempo de reconstrucción puede ser prolongado, y durante ese intervalo el sistema puede quedar más vulnerable a fallos adicionales. Esta situación se vuelve crítica cuando los discos son grandes y las cargas son constantes. Por ello, los sistemas contemporáneos combinan RAID con monitoreo, reemplazo preventivo, replicación distribuida, snapshots y políticas de recuperación más amplias.

La arquitectura distribuida transforma el almacenamiento masivo en un problema de coordinación entre nodos. En lugar de depender de un único servidor, los datos se fragmentan, replican y distribuyen entre múltiples máquinas. Esta organización permite escalar capacidad agregando nodos, mejorar throughput mediante acceso paralelo y tolerar fallos parciales sin detener el servicio completo. Qin (2025) plantea que la lógica central de una arquitectura distribuida consiste en dividir almacenamiento y cómputo entre nodos independientes que colaboran, lo que permite superar límites de hardware, sostener escalabilidad, mantener tolerancia a fallos y reducir ciclos de procesamiento mediante paralelismo.

Los sistemas de archivos distribuidos representan una pieza central de esta arquitectura. Su propósito es presentar una visión coherente del almacenamiento aunque los datos se encuentren físicamente repartidos. HDFS, CephFS, GlusterFS, PVFS y otros sistemas expresan diferentes formas de resolver el problema de almacenar grandes volúmenes de datos en múltiples nodos. Algunas arquitecturas se basan en esquemas maestro-esclavo, donde un nodo administra metadatos y otros almacenan bloques; otras distribuyen responsabilidades de forma más simétrica. Cada modelo introduce ventajas y limitaciones. Los enfoques centralizados simplifican coordinación y metadatos, pero pueden generar puntos de congestión o dependencia; los enfoques distribuidos aumentan disponibilidad y escalabilidad, pero incrementan complejidad de consistencia y administración.

HDFS constituye un ejemplo representativo de almacenamiento distribuido orientado a grandes volúmenes. Su diseño fragmenta archivos en bloques y los

replica en distintos nodos, lo que permite tolerar fallos y favorecer procesamiento cercano a los datos. Gunarathne, Wu, Qiu y Fox (2010) explican que Hadoop utiliza HDFS como sistema de archivos paralelo distribuido sobre clústeres de hardware común, logra confiabilidad mediante replicación de archivos y optimiza la comunicación al programar cómputos cerca de los datos usando la información de localidad proporcionada por el sistema de archivos. Esta estrategia es crucial porque reduce transferencias innecesarias y convierte la ubicación de los datos en una variable de planificación del procesamiento.

La localidad de datos es uno de los principios más importantes del almacenamiento masivo en Big Data. Si los datos se encuentran distribuidos, moverlos constantemente hacia nodos de cómputo remotos puede saturar la red y aumentar latencia. Una alternativa más eficiente consiste en mover la tarea hacia el nodo donde los datos ya residen, siempre que la disponibilidad de recursos lo permita. Esta lógica invierte la intuición clásica de procesamiento centralizado: no se trata de traer todos los datos al procesador, sino de asignar procesamiento donde la transferencia sea menor. En escalas masivas, esta decisión puede marcar la diferencia entre un sistema viable y uno dominado por tráfico de red.

Las bases NoSQL responden a otra dimensión del almacenamiento masivo: la diversidad estructural de los datos. Los modelos relacionales tradicionales son eficaces para datos altamente estructurados y transacciones consistentes, pero pueden resultar menos flexibles ante registros semiestructurados, documentos, grafos, series temporales, claves-valor y flujos de alta velocidad. Qin (2025) señala que las bases NoSQL se presentan como complemento de los sistemas relacionales porque ofrecen modelos flexibles y alta escalabilidad, especialmente en escenarios de escritura concurrente y datos heterogéneos. Esta flexibilidad no está exenta de costo, porque muchos sistemas NoSQL relajan consistencia inmediata para favorecer disponibilidad y partición, lo que obliga a evaluar cuidadosamente los requisitos de cada aplicación.

El almacenamiento masivo también exige mecanismos de tolerancia a fallos que operen más allá de la redundancia física. En sistemas distribuidos, los fallos no son eventos excepcionales, sino condiciones esperables. Un nodo puede

desconectarse, un disco puede fallar, una red puede interrumpirse, un proceso puede quedar rezagado o un centro de datos puede experimentar degradación parcial. Por ello, las arquitecturas modernas incorporan heartbeats, réplicas, reejecución de tareas, recuperación por linaje, snapshots, logs, checkpoints y balanceo de carga. Gunarathne et al. (2010) explican que Hadoop maneja fallos mediante reejecución de tareas fallidas y duplicación de tareas lentas, mientras conserva planificación basada en localidad de datos. Esta lógica evidencia que la disponibilidad depende tanto del almacenamiento como del motor de procesamiento que interactúa con él.

La replicación es una técnica central en sistemas de alta disponibilidad, pero su aplicación requiere equilibrio. Más réplicas aumentan tolerancia a fallos y pueden mejorar lectura distribuida, pero consumen más espacio y complejizan consistencia. Menos réplicas reducen costo, pero incrementan riesgo de pérdida o indisponibilidad. Algunos sistemas ajustan la cantidad de réplicas según criticidad, temperatura o frecuencia de acceso de los datos. Los datos calientes pueden requerir más copias para soportar alta demanda y reducir latencia; los datos fríos pueden mantenerse con menor redundancia si su acceso es ocasional y su recuperación tolera mayor demora. Este enfoque muestra que la disponibilidad eficiente no consiste en replicar todo por igual, sino en distribuir protección según valor, uso y riesgo.

La relación entre almacenamiento masivo y configuraciones RAID puede entenderse como una continuidad de escalas. RAID protege frente a fallos dentro de un arreglo local de discos; los sistemas distribuidos protegen frente a fallos de nodos, racks o clústeres; las arquitecturas cloud replican datos entre zonas o regiones; y los sistemas de Big Data agregan políticas de ubicación, partición, consistencia y recuperación. En todos los casos, el objetivo es sostener disponibilidad y rendimiento bajo condiciones de falla, pero los mecanismos se adaptan a la escala. A menor escala, la paridad y el espejado pueden ser suficientes; a mayor escala, se requieren replicación distribuida, tolerancia a particiones, balanceo, reconstrucción y administración de metadatos.

El almacenamiento masivo también está atravesado por el problema del desbordamiento. Incluso en arquitecturas in-memory, los datos pueden exceder

la memoria disponible y obligar a utilizar niveles persistentes. Esta situación exige decidir qué datos permanecen en memoria, cuáles se escriben en disco, cuáles se comprimen y cuáles se mueven a almacenamiento remoto. En aplicaciones iterativas o de análisis repetitivo, mantener conjuntos activos en memoria puede reducir significativamente el costo de acceso; en datos fríos, la persistencia en niveles inferiores puede ser suficiente. El reto consiste en diseñar políticas que distingan entre datos de trabajo, datos de respaldo, datos históricos, datos temporales y datos críticos.

Los modelos basados en RDD ilustran una estrategia intermedia entre memoria, almacenamiento y tolerancia a fallos. El sistema puede mantener datos en memoria para acelerar iteraciones, pero conserva información de linaje que permite recomputar particiones perdidas sin replicar todo de manera inmediata. Esta solución reduce ciertos costos de checkpointing, aunque puede aumentar el tiempo de recuperación si el linaje es largo o la recomputación es costosa. De esta manera, la disponibilidad no depende únicamente de copias físicas, sino también de la capacidad lógica de reconstruir datos a partir de transformaciones previas. Esta idea amplía el concepto clásico de respaldo: un dato puede protegerse mediante réplica, paridad, registro, snapshot o reconstrucción computacional.

En los sistemas científicos y de visualización, el almacenamiento masivo debe responder a flujos que no siempre se ajustan a consultas tradicionales. Simulaciones numéricas, datos temporales, imágenes volumétricas, sensores o resultados intermedios pueden generar archivos enormes que deben ser filtrados, comprimidos o priorizados. Cao et al. (2011) proponen una estrategia de compresión inteligente para datos volumétricos variables en el tiempo, orientada a reducir la cantidad de información transferida y mantener las características relevantes para el análisis científico. Este enfoque es importante porque demuestra que la gestión de almacenamiento no siempre consiste en guardar o mover todo, sino en seleccionar, reducir y preservar aquello que tiene mayor valor analítico.

En consecuencia, el almacenamiento masivo en la era del Big Data debe diseñarse como una arquitectura de decisiones. Cada dato debe situarse en función de su uso, tamaño, criticidad, frecuencia de acceso, necesidad de persistencia,

requerimiento de replicación, costo de transferencia y sensibilidad a latencia. Las configuraciones RAID aportan principios de distribución y redundancia; los sistemas distribuidos aportan escalabilidad y tolerancia a fallos; las bases NoSQL aportan flexibilidad de modelo; HDFS y sistemas similares aportan procesamiento cercano al dato; los sistemas cloud aportan elasticidad; y las técnicas de compresión o selección reducen presión sobre red y almacenamiento. El desafío consiste en integrar estas estrategias sin perder coherencia, disponibilidad ni eficiencia.

La gestión del ciclo de vida de los datos amplía todavía más la comprensión del almacenamiento masivo. En un sistema de Big Data, los datos no permanecen estáticos después de ser almacenados; atraviesan fases de generación, ingestión, registro, modelado, ubicación, procesamiento, análisis, visualización, archivo y eventual eliminación. Esta trayectoria exige que el almacenamiento dialogue con metadatos, políticas de seguridad, costos de transferencia, restricciones de privacidad, requisitos de disponibilidad y patrones de acceso. Mazumdar, Seybold, Kritikos y Verginadis (2019) plantean que los modelos de ciclo de vida de datos deben considerar actividades como colección, preparación, análisis y acción, además de la creación de valor a partir del conocimiento extraído, lo que evidencia que almacenar datos no es suficiente si no se administran sus condiciones de uso a lo largo del tiempo.

Los metadatos son fundamentales para que el almacenamiento masivo pueda operar con eficiencia. Sin información sobre ubicación, tamaño, formato, origen, frecuencia de acceso, permisos, número de réplicas, prioridad o dependencia respecto de aplicaciones, el sistema no puede decidir adecuadamente dónde colocar los datos ni cómo moverlos. En entornos cloud y multi-cloud, esta necesidad se vuelve más compleja porque los datos pueden estar distribuidos entre proveedores, regiones, servicios y niveles de almacenamiento. Mazumdar et al. (2019) sostienen que los metadatos permiten describir características estáticas y dinámicas de los datos, siendo una base para la gestión eficiente de almacenamiento, ubicación, migración y acceso en aplicaciones intensivas en datos.

La ubicación de datos, o data placement, se convierte entonces en una estrategia central de rendimiento y disponibilidad. Ubicar un dato no significa simplemente escoger un disco o un nodo disponible; implica anticipar dónde será procesado, cuántas veces será leído, qué latencia tolera la aplicación, qué costo tiene moverlo, qué nivel de confiabilidad requiere y qué restricciones normativas o institucionales condicionan su localización. En Big Data, una mala ubicación puede multiplicar el tráfico de red, aumentar tiempos de respuesta, saturar enlaces y elevar costos de operación. Por el contrario, una ubicación consciente de los patrones de acceso puede reducir transferencias, mejorar throughput y facilitar recuperación ante fallos. Esta lógica conecta almacenamiento, red y cómputo en una misma arquitectura de decisión.

Los sistemas distribuidos modernos también deben considerar el equilibrio entre consistencia, disponibilidad y tolerancia a particiones. En escenarios donde los datos están replicados entre nodos o regiones, una actualización debe propagarse para evitar divergencias. Sin embargo, exigir consistencia fuerte en todo momento puede aumentar latencia o reducir disponibilidad ante fallos de red. Relajar la consistencia puede mejorar escalabilidad y respuesta, pero introduce períodos en los que diferentes réplicas no reflejan exactamente el mismo estado. Este dilema afecta especialmente a bases NoSQL, sistemas geodistribuidos, servicios de nube y plataformas de alta concurrencia. Por ello, la selección de un sistema de almacenamiento debe partir de la naturaleza de la aplicación: no es igual un sistema financiero transaccional, un repositorio científico, una red social, un motor de recomendación o una plataforma de sensores.

Las configuraciones RAID y los sistemas distribuidos también enfrentan el problema de reconstrucción. Cuando un disco o nodo falla, los datos deben recuperarse desde paridad, réplicas o linaje. Durante ese proceso, el sistema consume ancho de banda, procesamiento y operaciones de lectura/escritura, lo que puede afectar el rendimiento normal. Además, si la reconstrucción tarda demasiado, el sistema permanece más expuesto a fallos adicionales. En arreglos de discos de gran capacidad, esta situación es especialmente crítica, porque el volumen a reconstruir puede ser enorme. Por ello, los sistemas modernos buscan estrategias como reconstrucción gradual, reconstrucción priorizada, codificación

de borrado, replicación selectiva, balanceo de carga y aislamiento de tráfico de recuperación.

La codificación de borrado aparece como una alternativa a la replicación completa cuando se busca reducir sobrecarga de almacenamiento. En lugar de guardar copias completas, los datos se dividen y se codifican en fragmentos redundantes que permiten reconstruir la información ante la pérdida de algunos de ellos. Esta estrategia puede mejorar eficiencia espacial, pero suele introducir mayor complejidad computacional y costos de reconstrucción. En escenarios de almacenamiento frío o de archivo, puede ser adecuada; en datos calientes con acceso frecuente o baja latencia, la replicación directa puede resultar más conveniente. Esto demuestra que no hay una política universal de disponibilidad. Cada técnica debe evaluarse por frecuencia de acceso, criticidad, costo de almacenamiento, tiempo de recuperación y nivel de riesgo aceptable.

La seguridad y la privacidad también forman parte del almacenamiento masivo en Big Data. A mayor distribución de datos, mayor superficie de exposición. Los datos pueden circular entre nodos, ser replicados en distintas ubicaciones, migrar entre nubes o permanecer en dispositivos de borde. Por ello, el almacenamiento debe integrar mecanismos de control de acceso, cifrado, auditoría, aislamiento, gestión de claves y cumplimiento normativo. La disponibilidad no puede perseguirse sacrificando confidencialidad o integridad. Un sistema altamente disponible pero vulnerable a accesos indebidos no satisface los requisitos de una arquitectura confiable. De ahí que la gestión de datos masivos deba integrar rendimiento, seguridad y gobernanza dentro de la misma estrategia de almacenamiento.

La presión del Big Data también ha impulsado arquitecturas híbridas de almacenamiento, en las que conviven HDD, SSD, NVM, memoria principal, almacenamiento en nube y repositorios distribuidos. Los datos se ubican según su temperatura: los datos calientes se mantienen en niveles rápidos; los datos tibios pueden alojarse en SSD o almacenamiento intermedio; los datos fríos se desplazan hacia discos de mayor capacidad o almacenamiento de archivo. Esta estratificación permite reducir costos sin sacrificar el rendimiento de las operaciones críticas. No obstante, requiere mecanismos capaces de detectar

cambios en los patrones de acceso, mover datos automáticamente y evitar que las migraciones generen más costo que beneficio.

En sistemas de alta disponibilidad, el almacenamiento también debe coordinarse con la recuperación de aplicaciones. No basta con que los datos existan; deben poder restaurarse en un estado consistente y útil. Los mecanismos de logging, checkpointing, snapshots y versionamiento permiten regresar a puntos anteriores, reconstruir estados y mitigar efectos de fallos o errores humanos. En sistemas in-memory, esta dimensión es crítica porque la volatilidad de la DRAM exige respaldos persistentes. Zhang et al. (2015) explican que, en sistemas in-memory, la tolerancia a fallos requiere mecanismos como logging, checkpointing, réplicas y recuperación rápida para garantizar durabilidad y consistencia transaccional ante fallas de energía, software o hardware.

La relación entre almacenamiento y procesamiento se vuelve especialmente importante en motores como Hadoop, Spark y plataformas distribuidas de análisis. En estas arquitecturas, el almacenamiento no solo conserva datos, sino que informa al planificador dónde ejecutar tareas para reducir movimiento. Spark, por ejemplo, busca mantener datos intermedios en memoria cuando resulta conveniente, pero puede recurrir a almacenamiento persistente o recomputación según la estrategia de tolerancia a fallos. Hadoop, por su parte, se apoya en HDFS y planificación cercana a datos. Estas soluciones evidencian que el almacenamiento masivo no debe diseñarse separado del motor de cómputo. La eficiencia surge cuando ambos niveles comparten información sobre localidad, carga, disponibilidad y dependencia de datos.

En los sistemas contemporáneos, las configuraciones RAID pueden interpretarse como una lógica fundacional de redundancia que luego se amplía en arquitecturas distribuidas. RAID enseña que la disponibilidad se construye mediante distribución, duplicación o paridad; Big Data extiende esos principios hacia clústeres, centros de datos y nubes. Sin embargo, la escala modifica el problema. En RAID, la unidad crítica suele ser el disco; en Big Data, pueden fallar nodos, redes, racks, servicios, zonas o regiones. Por ello, los sistemas de almacenamiento masivo combinan redundancia física, replicación lógica, control

de metadatos y políticas de ubicación. La confiabilidad emerge de la cooperación entre niveles, no de una técnica aislada.

La siguiente tabla sintetiza las principales estrategias de almacenamiento masivo y disponibilidad de datos que resultan pertinentes para sistemas de Big Data y alta disponibilidad. Su función es organizar comparativamente el aporte de cada técnica, sus ventajas, sus limitaciones y su uso preferente dentro de una arquitectura orientada a grandes volúmenes de información.

Tabla 5

Estrategias de almacenamiento masivo y disponibilidad de datos en Big Data

Estrategia o tecnología	Principio de funcionamiento	Ventaja principal	Limitación crítica	Uso recomendado en sistemas de alta disponibilidad
RAID 0	Distribución de datos en varios discos mediante striping	Mejora throughput y uso agregado de discos	No ofrece tolerancia a fallos	Cargas temporales o datos reconstruibles donde prime rendimiento
RAID 1	Replicación completa de datos entre discos	Alta disponibilidad local y recuperación simple	Alto costo de capacidad por duplicación	Datos críticos en servidores locales o sistemas que requieren recuperación rápida
RAID 5 / RAID 6	Paridad distribuida para recuperación ante fallos	Equilibrio entre capacidad y tolerancia a fallos	Penalización en escritura y reconstrucción costosa	Almacenamiento local con necesidad de redundancia moderada
RAID 10	Combinación de espejado y striping	Buen equilibrio entre rendimiento y tolerancia a fallos	Mayor costo de discos que RAID con paridad	Bases de datos, servicios transaccionales y cargas con alta demanda de E/S
HDFS y sistemas distribuidos de archivos	División de archivos en bloques replicados entre nodos	Escalabilidad, tolerancia a fallos y localidad de datos	Dependencia de metadatos y sobrecosto de replicación	Procesamiento batch, analítica distribuida y repositorios de Big Data
Bases NoSQL distribuidas	Modelos flexibles con partición y replicación	Escalabilidad horizontal y soporte para datos heterogéneos	Consistencia variable según diseño del sistema	Alta concurrencia, datos semiestructurados, documentos, clave-valor o grafos
Replicación geodistribuida	Copias en zonas o regiones diferentes	Continuidad operativa ante fallos regionales	Mayor latencia, costo y complejidad de consistencia	Servicios críticos, plataformas cloud y sistemas globales
Codificación de borrado	Fragmentación y codificación redundante de datos	Menor sobrecarga espacial que replicación completa	Reconstrucción más compleja y costosa	Almacenamiento frío, archivo y datos de acceso menos frecuente

Snapshots y checkpointing	Captura de estados consistentes para recuperación	Permite restaurar sistemas ante fallos o errores	Consume espacio y requiere coordinación temporal	Bases de datos, sistemas in-memory, HPC y servicios transaccionales
Data placement inteligente	Ubicación según patrones de acceso, costo y latencia	Reduce tráfico, mejora rendimiento y optimiza recursos	Requiere metadatos, monitoreo y adaptación continua	Cloud-Big Data, multi-cloud, edge y aplicaciones intensivas en datos

Nota. *Elaboración propia a partir de Zhang et al. (2015), Mazumdar et al. (2019), Qin (2025), Gunarathne et al. (2010), Cao et al. (2011) y Xie (2008).*

La tabla evidencia que la alta disponibilidad no depende de una única técnica, sino de una composición estratégica. RAID puede proteger frente a fallos locales; HDFS y sistemas distribuidos permiten escalar y sostener fallos de nodos; NoSQL aporta flexibilidad para datos heterogéneos; la replicación geodistribuida mejora continuidad operativa; la codificación de borrado reduce sobrecarga de almacenamiento; los snapshots y checkpoints facilitan recuperación; y el data placement inteligente busca que el dato esté en el lugar más conveniente según su uso. En conjunto, estas estrategias muestran que almacenar datos en la era del Big Data implica diseñar una arquitectura de resiliencia, no simplemente acumular capacidad.

El almacenamiento masivo también redefine el concepto de rendimiento. En los niveles superiores de memoria, el rendimiento suele asociarse con latencia de acceso y tasa de aciertos; en almacenamiento masivo, se relaciona con throughput de lectura/escritura, operaciones de entrada/salida por segundo, paralelismo de acceso, tiempo de recuperación, disponibilidad ante fallos, costo por terabyte y eficiencia de transferencia. Una arquitectura puede tener gran capacidad, pero bajo rendimiento si centraliza metadatos, concentra tráfico o no explota paralelismo. También puede tener excelente throughput, pero baja disponibilidad si carece de redundancia. En consecuencia, el rendimiento del almacenamiento debe medirse junto con confiabilidad, escalabilidad y costo operativo.

En entornos científicos, esta relación resulta decisiva. Las simulaciones de gran escala generan datos que deben almacenarse, analizarse, visualizarse y eventualmente compartir o archivar. Si el sistema de almacenamiento no

acompaña la velocidad de generación, el cálculo puede detenerse esperando escritura. Si no ofrece lectura suficiente, el análisis posterior se retrasa. Si no dispone de políticas de reducción, compresión o selección, los costos de conservación pueden crecer sin control. Por ello, la arquitectura de almacenamiento debe integrarse desde la planificación del flujo científico, no añadirse al final como repositorio. La ciencia intensiva en datos exige pensar simultáneamente cálculo, memoria, interconexión y persistencia.

Una estrategia óptima de almacenamiento para alta disponibilidad debe partir de la clasificación de los datos. No todos los datos tienen el mismo valor, frecuencia de acceso o sensibilidad. Los datos operativos críticos pueden requerir replicación fuerte y recuperación rápida; los datos históricos pueden tolerar mayor latencia; los datos derivados pueden reconstruirse; los datos únicos o irrepetibles requieren protección reforzada; los datos temporales pueden eliminarse después de su uso. Esta clasificación permite evitar la sobreprotección indiscriminada, que eleva costos, y la subprotección peligrosa, que compromete continuidad e integridad. La gestión eficiente se fundamenta en asignar niveles de protección según función y riesgo.

También debe considerarse la automatización. En sistemas pequeños, las políticas de almacenamiento pueden configurarse manualmente; en Big Data, la escala vuelve inviable esa administración artesanal. Se requieren mecanismos que monitoreen uso, detecten datos calientes, redistribuyan carga, ajusten réplicas, migren información, activen compresión, generen snapshots y respondan a fallos. Esta automatización no elimina la necesidad de diseño, sino que la desplaza hacia reglas, políticas y modelos de decisión. La arquitectura debe definir criterios para que el sistema actúe de forma adaptativa sin comprometer consistencia, seguridad ni eficiencia.

En síntesis, los sistemas de almacenamiento masivo y las configuraciones RAID muestran que la persistencia de datos en la era del Big Data es un problema arquitectónico de alta complejidad. Guardar información ya no significa depositarla en un medio físico, sino organizarla dentro de una estructura de redundancia, distribución, acceso, recuperación y gobierno. La disponibilidad se construye mediante múltiples capas: redundancia local, replicación distribuida,

protección por paridad, ubicación inteligente, metadatos, tolerancia a fallos, recuperación y monitoreo. Este enfoque permite comprender que la gestión de datos en sistemas modernos exige una visión integral donde almacenamiento, memoria, interconexión y procesamiento se diseñen como partes de una misma infraestructura de rendimiento y confiabilidad.

Discusión crítica

El contraste entre la velocidad de procesamiento y la velocidad de acceso a memoria constituye una de las tensiones más persistentes de la arquitectura computacional contemporánea. Durante décadas, la mejora del rendimiento se asoció con procesadores más rápidos, mayor frecuencia de reloj, más núcleos y unidades de ejecución más eficientes. Sin embargo, la experiencia acumulada en sistemas de alto rendimiento, Big Data y arquitecturas heterogéneas demuestra que el incremento de capacidad de cómputo no produce beneficios proporcionales cuando los datos no pueden moverse, localizarse o recuperarse al mismo ritmo. Esta brecha no es un detalle secundario de implementación, sino una condición estructural que determina la eficiencia real de los algoritmos científicos. Dally (1990) ya advertía que muchas aplicaciones concurrentes se encuentran limitadas por comunicación más que por procesamiento, especialmente cuando el tamaño de grano disminuye y los mensajes entre nodos se vuelven más frecuentes. Esta observación conserva plena vigencia en la era del Big Data, porque los sistemas actuales no solo calculan más, sino que comunican, transfieren, replican y sincronizan volúmenes de información cada vez mayores.

La memoria y la interconexión revelan que el rendimiento computacional no se agota en la capacidad aritmética. Un procesador puede ejecutar billones de operaciones por segundo, pero si los operandos permanecen en memoria principal, almacenamiento remoto o dispositivos externos sin suficiente ancho de banda, la unidad de cálculo queda subutilizada. Esta situación obliga a revisar la manera en que se diseñan los algoritmos científicos. La eficiencia algorítmica tradicional, centrada en el número de operaciones, resulta incompleta si no incorpora el costo del movimiento de datos. Un algoritmo con menor complejidad teórica puede comportarse peor que otro más costoso en operaciones si produce accesos irregulares, fallos de caché, comunicación excesiva o transferencias host-

dispositivo no amortizadas. Desde esta perspectiva, la arquitectura de memoria introduce un criterio de evaluación que transforma la noción misma de eficiencia: no basta con calcular menos, también es necesario mover menos y reutilizar mejor.

La jerarquía de memoria aparece como una solución parcial frente a esta brecha, pero también como un recordatorio de su profundidad. Los registros, cachés, DRAM, memorias no volátiles y almacenamiento masivo conforman una arquitectura de proximidades diseñada para ocultar latencia y explotar localidad. No obstante, esta solución depende de que las aplicaciones presenten patrones de acceso compatibles con la organización jerárquica. Cuando los datos se acceden de forma secuencial, bloqueada o con reutilización temporal, la jerarquía puede ofrecer beneficios significativos; cuando los accesos son aleatorios, dispersos o dependientes de estructuras irregulares, los fallos de caché y accesos a niveles inferiores pueden dominar el tiempo de ejecución. Zhang, Chen, Ooi, Tan y Zhang (2015) señalan que los sistemas in-memory, aunque reducen el cuello de botella del disco, se vuelven más sensibles a la utilización de caché, la organización de datos, el paralelismo y el control de concurrencia. Esto muestra que mover los datos desde disco hacia memoria no elimina la brecha, sino que desplaza la presión hacia niveles más finos de la jerarquía.

La tensión se profundiza en las arquitecturas heterogéneas. Las GPU, FPGA y aceleradores especializados ofrecen gran capacidad de cómputo, pero suelen depender de transferencias costosas desde y hacia la memoria del host o desde otros niveles de almacenamiento. En estos casos, la aceleración solo es efectiva cuando el volumen de cálculo compensa el costo de comunicación. Si una tarea exige transferencias frecuentes de pequeños bloques, sincronización constante o retorno inmediato de resultados, el bus o la interconexión puede cancelar gran parte de la ganancia computacional. Yellapragada (2022) muestra que en programas acelerados el flujo típico comienza y termina con movimientos de datos entre CPU y GPU, por lo que la superposición entre comunicación y cómputo se vuelve indispensable para mejorar rendimiento. Esta evidencia confirma que el diseño de algoritmos científicos debe incorporar explícitamente la ubicación de los datos, la granularidad de las transferencias y la posibilidad de solapar operaciones.

La brecha entre cómputo y acceso también condiciona la elección de estructuras de datos. En sistemas intensivos en memoria, una estructura conceptualmente elegante puede resultar ineficiente si produce saltos frecuentes, punteros dispersos o baja localidad espacial. Por el contrario, estructuras compactas, alineadas y orientadas a bloques pueden mejorar significativamente el rendimiento aunque parezcan menos expresivas desde el punto de vista abstracto. Esta tensión afecta bases de datos, simulaciones numéricas, grafos, modelos de aprendizaje automático y visualización científica. Kozar et al. (2025) destacan que los procesadores heterogéneos presentan diferencias arquitectónicas relevantes, pues las CPU favorecen ciertos patrones de memoria secuencial mientras que las GPU obtienen mejores resultados con accesos coalescentes y paralelismo masivo. Por ello, un algoritmo portable no es necesariamente un algoritmo eficiente en todas las arquitecturas.

El problema de la memoria también introduce una dimensión energética. Mover datos consume energía, y en muchos sistemas modernos el costo energético de transferir información entre niveles de memoria puede superar el costo de operar sobre ella. Esta situación tiene implicaciones para centros de datos, HPC y sistemas de Big Data, donde la eficiencia energética es una condición de sostenibilidad técnica y económica. La arquitectura no puede perseguir únicamente más cómputo, porque cada transferencia innecesaria incrementa consumo, calor, congestión y costo operativo. Vaithianathan (2025) plantea que la optimización de la jerarquía de memoria en HPC debe considerar estrategias como reconfiguración dinámica, tecnologías emergentes y jerarquías sensibles al comportamiento de la aplicación, precisamente porque el acceso a memoria afecta simultáneamente rendimiento y eficiencia. En términos críticos, el problema no es solo que la memoria sea más lenta que el procesador, sino que mover datos de manera ineficiente vuelve insostenible el escalamiento.

Las redes de interconexión amplifican esta problemática en sistemas distribuidos. Cuando los datos se reparten entre nodos, racks o centros de datos, la latencia ya no depende únicamente de la jerarquía interna de una máquina, sino de topologías, rutas, congestión, distancia física, bisección, protocolos y planificación de tareas. Agarwal (1991) argumenta que la latencia de una red depende tanto de parámetros estructurales, como dimensión, ancho de canal y

retardos de nodo o cable, como de los patrones de comunicación de las aplicaciones. Esto significa que un algoritmo científico no puede analizarse de manera independiente de la red donde se ejecuta. Una simulación con comunicación entre vecinos puede beneficiarse de topologías que preserven localidad; un algoritmo con reducciones globales frecuentes puede quedar limitado por sincronización; y una aplicación de análisis distribuido puede degradarse si los datos se ubican lejos de las tareas que los consumen.

La ubicación de datos se convierte, por tanto, en una forma de optimización algorítmica. En Big Data, no siempre conviene mover grandes conjuntos hacia el nodo más potente; muchas veces resulta más eficiente desplazar el cómputo hacia donde los datos ya residen. Esta idea redefine la relación entre almacenamiento y procesamiento. Los sistemas de archivos distribuidos, las bases NoSQL, las arquitecturas cloud y los motores de análisis intentan explotar localidad para reducir tráfico y mejorar throughput. Mazumdar, Seybold, Kritikos y Verginadis (2019) enfatizan que la ubicación de datos en ecosistemas cloud-Big Data debe considerar latencias, patrones de acceso, costos y requisitos no funcionales, lo que evidencia que el almacenamiento no es solo una decisión de capacidad, sino una decisión de rendimiento. El algoritmo científico contemporáneo debe, por tanto, ser consciente de la geografía de los datos.

La existencia de RAID, replicación, snapshots, checkpoints y codificación de borrado demuestra que la disponibilidad tiene un costo. Proteger datos implica duplicar, calcular paridad, registrar cambios, mantener metadatos, reconstruir estados y consumir recursos de almacenamiento y comunicación. Esta situación plantea una tensión entre rendimiento y confiabilidad. Un sistema sin redundancia puede ser rápido, pero frágil; uno excesivamente replicado puede ser confiable, pero costoso y lento en escrituras o sincronización. En sistemas científicos, esta tensión es particularmente delicada porque la pérdida de datos puede comprometer experimentos extensos, simulaciones costosas o resultados irrepetibles. En consecuencia, el diseño de almacenamiento debe equilibrar la necesidad de proteger información con la necesidad de sostener flujos de análisis eficientes.

También existe una tensión entre abstracción y materialidad. Los modelos de programación suelen ofrecer al investigador la ilusión de que los datos están disponibles en estructuras lógicas uniformes: arreglos, matrices, tablas, grafos, colecciones distribuidas o dataframes. Sin embargo, esas abstracciones se materializan en cachés, páginas, bloques, particiones, réplicas, enlaces y dispositivos con velocidades heterogéneas. Cuando el investigador ignora esa materialidad, puede diseñar algoritmos correctos pero ineficientes. Cuando la considera en exceso, puede producir soluciones altamente optimizadas pero poco portables o difíciles de mantener. El desafío contemporáneo consiste en construir abstracciones que no oculten completamente el costo del dato, sino que permitan al programador razonar sobre localidad, transferencia y persistencia sin quedar atrapado en detalles de bajo nivel.

Las tecnologías ópticas y las topologías de baja latencia ofrecen caminos prometedores, pero no eliminan la necesidad de un diseño algorítmico consciente. Shacham y Bergman (2007) muestran que las redes fotónicas pueden reducir ciertas limitaciones de las interconexiones eléctricas y ofrecer rutas de baja latencia, aunque también enfrentan desafíos como la ausencia de buffers ópticos eficientes. Proietti, Cao, Nitta, Li y Yoo (2015) proponen interconexiones ópticas jerárquicas capaces de mejorar escalabilidad y throughput en sistemas de gran tamaño. Sin embargo, incluso con enlaces más rápidos, los algoritmos seguirán enfrentando costos de sincronización, coordinación, consistencia y ubicación. La tecnología de interconexión puede ampliar el margen de rendimiento, pero no reemplaza la necesidad de reducir comunicación innecesaria ni de organizar datos de forma adecuada.

La confiabilidad de la memoria introduce otra limitación crítica. A medida que las tecnologías DRAM se escalan, aumentan los desafíos asociados con errores, corrección, refresco y caracterización de fallos. Patel, Kim, Hassan y Mutlu (2019) sostienen que los mecanismos de ECC incorporados en DRAM moderna pueden ocultar las distribuciones reales de error, dificultando el estudio directo del comportamiento físico de los dispositivos. Esta situación tiene una implicación relevante: la memoria no es solo lenta respecto del procesador, sino también físicamente vulnerable. Los algoritmos científicos que operan sobre grandes volúmenes de datos durante largos períodos dependen de infraestructuras

capaces de preservar integridad. La velocidad sin confiabilidad puede producir resultados incorrectos con apariencia de precisión.

En este marco, los algoritmos científicos deben evolucionar hacia una lógica data-aware. Esto implica diseñar procedimientos que no solo minimicen operaciones, sino que organicen el acceso a memoria, reduzcan transferencias, exploten localidad, agrupen comunicaciones, eviten sincronizaciones innecesarias, usen compresión selectiva, consideren tolerancia a fallos y se adapten a la arquitectura de ejecución. Un algoritmo científico moderno debería responder preguntas que antes pertenecían exclusivamente al hardware o al sistema operativo: dónde están los datos, cuántas veces se reutilizarán, cuánto cuesta moverlos, qué nivel de memoria conviene utilizar, qué partes pueden ejecutarse cerca del dato, qué información debe replicarse y qué resultados pueden recomputarse.

Esta transformación tiene consecuencias educativas. Enseñar arquitectura y organización de computadores en la era del Big Data exige superar una visión fragmentada de CPU, memoria, buses y almacenamiento como temas separados. El estudiante debe comprender que todos estos elementos forman una cadena de dependencia. La latencia del bus afecta al acelerador; la caché afecta al algoritmo; la red afecta a la distribución; el almacenamiento afecta a la disponibilidad; la replicación afecta al costo; y la memoria afecta a la confiabilidad del resultado. La arquitectura deja de ser una colección de componentes para convertirse en una teoría aplicada del movimiento de datos.

Desde una perspectiva crítica, la brecha entre procesamiento y acceso a memoria también revela límites de la idea de progreso tecnológico lineal. No basta con fabricar procesadores más rápidos o memorias más grandes. Cada incremento de capacidad genera nuevas formas de cuello de botella. Más núcleos pueden incrementar contención de caché; más aceleradores pueden saturar PCIe; más memoria puede exigir mejor gestión NUMA; más almacenamiento puede alargar reconstrucciones; más nodos pueden aumentar tráfico y sincronización; más réplicas pueden complicar consistencia. La arquitectura computacional avanza resolviendo problemas y creando otros de escala superior. Por ello, la innovación no puede medirse solo por potencia agregada, sino por equilibrio sistémico.

La brecha entre velocidad de procesamiento y velocidad de acceso a memoria condiciona profundamente el diseño de algoritmos científicos. No se trata de una limitación técnica aislada, sino de un principio organizador del cómputo contemporáneo. El rendimiento efectivo depende de cómo se articulan registros, cachés, DRAM, almacenamiento, buses, redes, aceleradores y políticas de ubicación. Las arquitecturas modernas intentan reducir la distancia entre dato y cómputo mediante jerarquías, interconexiones rápidas, almacenamiento distribuido, procesamiento in-memory, aceleradores y data placement. No obstante, ninguna de estas estrategias elimina por completo el problema. La eficiencia científica en la era del Big Data exige comprender que calcular es inseparable de mover, proteger, ubicar y recuperar datos. Esa es la condición crítica que redefine el diseño arquitectónico y algorítmico de los sistemas computacionales actuales.

III. CONCLUSIONES

En conclusión, las dinámicas de interconexión y las jerarquías de memoria constituyen dimensiones esenciales para comprender el rendimiento de los sistemas computacionales en la era del Big Data. El capítulo ha mostrado que la velocidad de procesamiento, aunque relevante, no es suficiente para explicar la eficiencia real de una arquitectura. Los datos deben desplazarse entre registros, cachés, memoria principal, aceleradores, buses, redes, dispositivos de almacenamiento y sistemas distribuidos, y cada movimiento introduce costos de latencia, ancho de banda, energía, sincronización y confiabilidad. Por ello, la arquitectura contemporánea debe interpretarse como una organización integral del flujo de datos, donde el procesamiento depende de la capacidad del sistema para ubicar, transferir, proteger y reutilizar la información de manera oportuna.

Las estructuras de buses y los protocolos de arbitraje permiten reconocer que la comunicación interna del hardware es una zona crítica de coordinación. En sistemas simples, el bus compartido ofrece integración y bajo costo, pero en plataformas multiprocesador, heterogéneas o intensivas en datos puede convertirse en un punto de contención. La evolución hacia buses multicapa, enlaces especializados, redes en chip, interconexiones CPU-GPU, topologías de alto rendimiento y tecnologías ópticas responde a la necesidad de reducir esperas

y sostener mayor concurrencia. Sin embargo, ninguna interconexión es óptima por sí misma; su pertinencia depende del tipo de carga, del patrón de comunicación, de la escala del sistema, de la localidad de los datos y de los límites físicos de implementación.

La jerarquía de memoria, por su parte, expresa una solución arquitectónica basada en compromisos. Los registros y cachés ofrecen velocidad, pero capacidad reducida; la memoria principal permite sostener el trabajo activo del sistema, pero mantiene una brecha de latencia frente al procesador; las memorias no volátiles y el almacenamiento persistente amplían capacidad y durabilidad, aunque con mayores costos de acceso. Esta organización escalonada solo resulta eficiente cuando los datos se colocan en el nivel adecuado y cuando los algoritmos aprovechan localidad espacial y temporal. En consecuencia, la gestión de memoria no puede limitarse al hardware; debe articularse con estructuras de datos, compiladores, sistemas operativos, bases de datos, motores de procesamiento y decisiones algorítmicas.

El análisis también permitió evidenciar que la memoria moderna debe evaluarse no solo por velocidad, sino también por confiabilidad. La corrección de errores, la coherencia de caché, la integridad de DRAM, la recuperación ante fallos y la consistencia de datos son condiciones indispensables para que el rendimiento tenga valor real. Un sistema rápido pero incapaz de preservar la exactitud de la información compromete la validez de los resultados que produce. En contextos científicos, financieros, biomédicos, educativos o gubernamentales, la disponibilidad y la integridad son tan importantes como el tiempo de respuesta. Por ello, las arquitecturas eficientes deben integrar rendimiento y confiabilidad como dimensiones inseparables.

Los sistemas de almacenamiento masivo y las configuraciones RAID amplían esta reflexión hacia la persistencia de datos. El almacenamiento en la era del Big Data no consiste únicamente en acumular capacidad, sino en diseñar estrategias de redundancia, replicación, recuperación, ubicación y clasificación de datos. RAID aporta principios clásicos de distribución y protección local; los sistemas de archivos distribuidos, las bases NoSQL, la replicación geográfica, los snapshots, el checkpointing y el data placement inteligente extienden esos principios hacia

infraestructuras de gran escala. La alta disponibilidad se construye mediante capas complementarias, no mediante una técnica única. Su eficacia depende de clasificar los datos según criticidad, frecuencia de acceso, valor, riesgo y costo de recuperación.

Desde una perspectiva crítica, el capítulo permite afirmar que la brecha entre procesamiento y acceso a memoria condiciona profundamente el diseño de algoritmos científicos. La eficiencia ya no puede evaluarse solo por la cantidad de operaciones, sino también por la cantidad de datos movidos, la frecuencia de acceso, la localidad, la comunicación entre nodos, la transferencia hacia aceleradores, la persistencia y la recuperación. Un algoritmo moderno debe ser consciente de la arquitectura en la que se ejecuta. Debe minimizar transferencias innecesarias, favorecer accesos contiguos, explotar la reutilización de datos, reducir sincronizaciones, ubicar el procesamiento cerca de la información y considerar mecanismos de tolerancia a fallos desde su concepción.

Las estrategias óptimas para la gestión de datos en sistemas de alta disponibilidad deben orientarse hacia una visión integral. Es necesario combinar jerarquías de memoria sensibles a la carga, interconexiones de baja latencia, almacenamiento distribuido, redundancia ajustada al valor del dato, políticas inteligentes de ubicación, mecanismos de recuperación rápida y diseño conjunto entre hardware y software. La arquitectura computacional contemporánea debe avanzar hacia sistemas capaces de equilibrar velocidad, escalabilidad, confiabilidad, energía y costo. En la era del Big Data, el desafío no es solamente procesar más información, sino hacerlo de manera sostenible, segura, disponible y significativa, garantizando que los datos estén en el lugar correcto, en el momento correcto y con el nivel de protección adecuado para convertirse en conocimiento útil.

CAPITULO III: EL MICROPROCESADOR MODERNO: PARALELISMO Y SEGMENTACIÓN INSTRUCCIONAL

Chapter III: The Modern Microprocessor: Parallelism and Instructional Segmentation

CAPÍTULO III: EL MICROPROCESADOR MODERNO: PARALELISMO Y SEGMENTACIÓN INSTRUCCIONAL

Chapter III: The Modern Microprocessor: Parallelism and Instructional Segmentation

I. Introducción

El microprocesador moderno constituye el núcleo operativo de la computación contemporánea, porque concentra la ejecución lógica, aritmética y de control que permite transformar instrucciones codificadas en operaciones materiales sobre datos. Sin embargo, comprenderlo únicamente como una unidad central de procesamiento que recupera, decodifica y ejecuta instrucciones de manera secuencial sería insuficiente para explicar su complejidad actual. La CPU contemporánea es una microarquitectura profundamente estratificada, compuesta por unidades funcionales, registros, pipelines, predictores, memorias caché, unidades vectoriales, mecanismos de ejecución especulativa, control energético y, en muchos casos, múltiples núcleos integrados en un mismo chip. La evolución de los microprocesadores durante las últimas décadas muestra precisamente este desplazamiento: desde diseños centrados en una única CPU hacia sistemas multicore y manycore capaces de explotar diferentes formas de paralelismo para sostener el incremento de rendimiento ante los límites físicos de frecuencia, disipación térmica y complejidad de integración (Nikolić et al., 2022).

En su formulación básica, el microprocesador ejecuta un ciclo de instrucción que comprende la búsqueda de la instrucción, su decodificación, la obtención de operandos, la ejecución de la operación y la escritura del resultado. Esta secuencia expresa la lógica elemental del procesamiento programado, pero la arquitectura moderna no se limita a realizarla de manera lineal. La segmentación instruccional, o pipelining, fragmenta ese ciclo en etapas parcialmente superpuestas para que varias instrucciones se encuentren en diferentes fases de ejecución al mismo tiempo. La finalidad de esta técnica no es reducir necesariamente la latencia de una instrucción aislada, sino aumentar el throughput general del procesador. En una arquitectura segmentada, mientras

una instrucción se ejecuta en la ALU, otra puede estar siendo decodificada y otra recuperada desde memoria de instrucciones. Esta superposición convierte el tiempo secuencial en una estructura de flujo, pero también introduce riesgos derivados de dependencias de datos, conflictos por recursos y saltos de control.

La segmentación, por tanto, no debe interpretarse como una solución mecánica sin costos. Al dividir el procesamiento en etapas, el sistema debe garantizar que cada instrucción reciba datos correctos, que los recursos no sean utilizados simultáneamente por instrucciones incompatibles y que el flujo de ejecución no se vea interrumpido por decisiones de salto aún no resueltas. Los diseños RISC de cinco etapas, usualmente representados mediante las fases IF, ID, EX, MEM y WB, ofrecen un modelo pedagógico claro para comprender esta problemática, porque muestran cómo la búsqueda, decodificación, ejecución, acceso a memoria y escritura de resultados pueden organizarse en una tubería de procesamiento. No obstante, el mismo modelo evidencia que el rendimiento del pipeline depende de la capacidad del procesador para detectar y resolver hazards mediante forwarding, interlocks, stalls, flushes y lógica especializada de control (Ponrani et al., 2026).

Esta problemática se intensifica en los procesadores superscalar, cuyo objetivo es emitir y ejecutar varias instrucciones por ciclo mediante múltiples unidades funcionales. La arquitectura superscalar amplía el paralelismo a nivel de instrucción, pero también incrementa la complejidad de la dependencia entre instrucciones. Para que el procesador pueda sostener un alto número de instrucciones por ciclo, debe alimentar permanentemente el pipeline, seleccionar instrucciones disponibles, evitar conflictos, anticipar saltos y recuperar el estado correcto cuando una predicción falla. Pizzol, Pilla y Navaux (s. f.) muestran que la predicción de saltos se vuelve decisiva en procesadores superscalar, porque las dependencias de control reducen el tamaño efectivo de los bloques básicos y afectan el flujo de instrucciones hacia las etapas de ejecución. De esta manera, la CPU moderna no solo ejecuta instrucciones; también especula, reordena, predice, recupera y administra incertidumbre.

El problema de los saltos condicionales permite observar una de las tensiones más profundas del microprocesador moderno. La ejecución secuencial presupone

que el procesador conoce cuál será la siguiente instrucción; sin embargo, una rama condicional introduce una bifurcación cuyo resultado puede depender de cálculos aún no completados. Si el procesador espera hasta resolver el salto, pierde ciclos valiosos; si especula y se equivoca, debe descartar trabajo incorrecto y restaurar el estado apropiado. Por ello, la predicción de saltos se ha convertido en un componente central de la microarquitectura, especialmente en pipelines profundos y anchos. Zhou, Önder y Carr (2005) explican que, conforme aumentan la profundidad del pipeline y el tamaño de la ventana de instrucciones, la penalización por mala predicción de saltos se vuelve un factor crítico del rendimiento global, lo que ha impulsado mecanismos de recuperación más rápidos frente a mispredictions.

Junto con la predicción, la ejecución fuera de orden representa otra estrategia fundamental para sostener el paralelismo instruccional. En un procesador estrictamente en orden, una instrucción detenida por dependencia o acceso a memoria puede bloquear a las siguientes, aunque algunas de ellas estén listas para ejecutarse. La ejecución fuera de orden intenta evitar esa pérdida de oportunidad permitiendo que instrucciones independientes avancen antes, siempre que el resultado final preserve la semántica del programa. Para ello se requieren estructuras como renombramiento de registros, colas de emisión, estaciones de reserva, reorder buffer y mecanismos de commit que permitan distinguir entre ejecución interna y estado arquitectónico visible. Esta separación entre orden lógico del programa y orden físico de ejecución es uno de los rasgos más sofisticados de los microprocesadores modernos, porque permite extraer paralelismo oculto en secuencias aparentemente lineales.

La discusión sobre el microprocesador moderno también exige abordar la relación entre ISA y microarquitectura. Durante décadas, la comparación entre RISC y CISC fue presentada como una oposición entre instrucciones simples y complejas, longitud fija y variable, ejecución rápida y compactación de código. Sin embargo, la evidencia contemporánea muestra que esta dicotomía debe matizarse. Los procesadores x86 modernos pueden traducir instrucciones complejas en micro-operaciones internas más simples, mientras que muchos procesadores RISC incorporan mecanismos microarquitectónicos complejos para aumentar rendimiento, reducir energía o soportar paralelismo. En

consecuencia, las diferencias observadas entre plataformas no pueden atribuirse únicamente a la ISA, porque también intervienen la jerarquía de memoria, la organización de núcleos, la integración del sistema, la política de energía, el compilador y el tipo de carga de trabajo. Özyılmaz (2026) sostiene que las comparaciones entre ARM y x86-64 deben interpretarse como resultados de plataforma y no como efectos puros de la ISA, debido a la influencia conjunta de microarquitectura, memoria, organización de núcleos, integración y mecanismos de gestión energética.

Aun así, el debate RISC/CISC conserva relevancia porque permite comprender distintas filosofías de diseño. RISC privilegia regularidad, simplicidad de instrucción, modelos load-store y facilidad de segmentación; CISC favorece instrucciones más expresivas, compatibilidad histórica y densidad de código. En sistemas embebidos, IoT, edge computing y dispositivos de bajo consumo, esta discusión adquiere una importancia renovada porque la selección de la arquitectura afecta consumo energético, latencia de respuesta, soporte de herramientas, predictibilidad temporal y complejidad de implementación. Rosdiyanto, Zuhro y Nisfuwadi (2025) plantean que las arquitecturas RISC, en especial ARM Cortex y RISC-V, ofrecen ventajas de flexibilidad, modularidad y eficiencia energética en sistemas embebidos modernos, mientras que CISC conserva fortalezas asociadas a compatibilidad, riqueza semántica e integración con ecosistemas industriales tradicionales.

La emergencia de RISC-V amplía esta discusión al introducir una ISA abierta, modular y extensible. Su relevancia no se limita al hecho de ser royalty-free, sino a la posibilidad de adaptar el repertorio de instrucciones a dominios específicos, como procesamiento digital de señales, inteligencia artificial, control embebido o aplicaciones de bajo consumo. Las extensiones SIMD y, en particular, las propuestas orientadas a packed-SIMD, evidencian que la frontera entre ISA general y aceleración especializada se vuelve cada vez más flexible. Sultanpuri (2025) muestra que la implementación de la extensión P de RISC-V permite reducir ciclos de CPU e instrucciones retiradas en kernels DSP como producto punto, multiplicación de matrices, filtros FIR y detección de bordes Sobel, aunque los beneficios dependen del tamaño de la carga, del empaquetamiento de datos y de los límites del paralelismo disponible.

La unidad de control ocupa un lugar esencial en este escenario, porque traduce las instrucciones en señales que coordinan el datapath. En arquitecturas cableadas, la lógica de control se implementa mediante circuitos combinacionales y secuenciales diseñados para activar operaciones específicas. En arquitecturas microprogramadas, en cambio, la ejecución de instrucciones se organiza mediante microinstrucciones almacenadas en una memoria de control, lo que permite una forma más flexible de implementar comportamientos complejos. La microprogramación cumple así una función conceptual decisiva: muestra que entre la ISA visible y el hardware físico existe un nivel intermedio de control que puede diseñarse, simularse, enseñar y optimizarse. Heinikoski (2023) aborda este valor desde una perspectiva educativa al desarrollar un simulador web de una máquina microprogramada, orientado a facilitar la comprensión de registros, buses, memoria, ALU, reloj y microinstrucciones mediante ejecución visual e interactiva.

El análisis del microprocesador moderno debe incorporar también las restricciones energéticas. Durante una etapa importante de la historia de la computación, la mejora de rendimiento se asoció con el aumento de frecuencia y densidad de transistores. No obstante, la disipación de potencia, el calentamiento y los límites de escalamiento obligaron a desplazar el diseño hacia multicore, paralelismo, heterogeneidad, unidades especializadas y técnicas de ahorro energético. En los procesadores actuales, el rendimiento ya no puede separarse del consumo. Un diseño que ejecuta más instrucciones por ciclo, predice más agresivamente, mantiene ventanas de instrucciones más amplias o integra más unidades funcionales también puede incrementar área, complejidad y energía. Por ello, las microarquitecturas contemporáneas se diseñan bajo compromisos entre velocidad, potencia, área, latencia, throughput, confiabilidad y adaptabilidad.

Esta tensión entre rendimiento y energía se vuelve más visible en dominios como DSP, FPGA, FPU, sistemas embebidos y procesamiento específico de aplicación. Las unidades de punto flotante, los procesadores reconfigurables y las arquitecturas especializadas muestran que no siempre conviene utilizar una CPU general para todas las tareas. En ciertos escenarios, una unidad especializada puede lograr mejor rendimiento por watt o menor área para una operación

dominante. Sin embargo, la especialización reduce flexibilidad y exige herramientas de diseño, compiladores, verificación y soporte de software adecuados. Lee, Choi y Dutt (2003) proponen la síntesis de conjuntos de instrucciones específicos de aplicación para mejorar eficiencia energética en procesadores ASIP, mostrando que la personalización de instrucciones puede reducir energía y producto energía-retardo cuando se optimizan simultáneamente codificación, conteo dinámico de instrucciones y estructura del procesador.

En consecuencia, el microprocesador moderno puede interpretarse como un espacio de negociación entre varias racionalidades arquitectónicas. La segmentación busca mantener ocupadas las etapas de ejecución; el diseño superscalar intenta emitir múltiples instrucciones por ciclo; la predicción de saltos reduce esperas de control; la ejecución fuera de orden aprovecha instrucciones listas aunque el programa sea secuencial; RISC y CISC expresan distintas estrategias de codificación y ejecución; la microprogramación revela un nivel intermedio de control; las extensiones SIMD y los aceleradores acercan el hardware a dominios específicos; y las políticas energéticas intentan evitar que el rendimiento se alcance a costa de una disipación insostenible. Ninguna de estas dimensiones opera de manera aislada. Todas interactúan en el interior de una microarquitectura que debe producir resultados correctos, rápidos, eficientes y compatibles con el ecosistema de software.

Este capítulo se desarrolla a partir de cuatro ejes. Primero, se examina el ciclo de instrucción y la segmentación como fundamento del paralelismo instruccional, incluyendo hazards, forwarding, interlocks y técnicas de mitigación. Segundo, se analiza la comparación entre RISC y CISC desde una mirada contemporánea que incorpora ARM, x86-64, RISC-V, sistemas embebidos, eficiencia energética y plataformas reales. Tercero, se estudian las unidades de control microprogramadas como puente entre instrucciones visibles y señales físicas de ejecución. Cuarto, se abordan la predicción de saltos y la ejecución fuera de orden como mecanismos esenciales para sostener pipelines profundos, procesadores superscalar y ventanas de instrucciones amplias. Posteriormente, la discusión crítica problematiza el equilibrio entre eficiencia energética y rendimiento computacional en microarquitecturas de última generación.

Desde esta perspectiva, el microprocesador moderno no debe ser entendido como un simple heredero lineal de la CPU clásica, sino como una construcción compleja que integra historia, abstracción, circuitería, paralelismo, especulación, control y energía. Su refinamiento ha sido necesario para sostener aplicaciones de inteligencia artificial, computación científica, procesamiento digital de señales, control en tiempo real, sistemas embebidos y plataformas de alto rendimiento. Al mismo tiempo, esa complejidad introduce nuevas tensiones: mayor dificultad de verificación, costos de recuperación ante fallos especulativos, consumo energético creciente, dependencia de compiladores y pérdida de predictibilidad temporal en ciertos dominios. El estudio del microprocesador moderno exige, por tanto, una mirada integral capaz de articular la lógica del ciclo de instrucción con las estrategias microarquitectónicas que permiten, limitan y condicionan el rendimiento real de los sistemas computacionales actuales.

II. DESARROLLO

El ciclo de instrucción y la segmentación (pipelining)

El ciclo de instrucción constituye la unidad lógica básica mediante la cual un microprocesador transforma un programa almacenado en una secuencia de operaciones ejecutables. En su forma más elemental, este ciclo comprende la búsqueda de la instrucción en memoria, su decodificación, la localización de operandos, la ejecución de la operación requerida y la escritura del resultado en el destino correspondiente. Aunque esta descripción parece lineal, su importancia conceptual es profunda, porque permite comprender cómo el programa, entendido como secuencia simbólica de instrucciones, se convierte en actividad material dentro del datapath. El microprocesador no interpreta una instrucción como una entidad aislada, sino como un conjunto de señales, movimientos internos, accesos a registros, operaciones de la ALU, posibles accesos a memoria y actualizaciones del estado arquitectónico. En este sentido, el ciclo de instrucción revela la articulación entre ISA, unidad de control, registros, memoria y organización física del procesador.

En los microprocesadores tempranos, las fases del ciclo de instrucción se ejecutaban predominantemente de manera secuencial. La instrucción debía ser

buscada, decodificada y ejecutada antes de iniciar plenamente el procesamiento de la siguiente. Esta organización simplificaba el control, pero desaprovechaba recursos internos durante gran parte del tiempo, porque mientras una etapa realizaba su trabajo, otras permanecían inactivas. La segmentación instruccional surgió como respuesta a esta ineficiencia, al dividir el ciclo de instrucción en etapas sucesivas que pueden operar de manera parcialmente simultánea sobre instrucciones diferentes. Nikolić, Dimitrijević, Nikolić y Stojčev (2022) explican que, desde la segunda generación de microprocesadores, la superposición de fases como búsqueda, decodificación y ejecución permitió aumentar la velocidad de procesamiento respecto de los diseños seriales iniciales, anticipando la consolidación posterior del pipelining como técnica básica de mejora del rendimiento.

La lógica del pipelining puede entenderse mediante una analogía productiva: en lugar de esperar a que una instrucción complete todas sus fases para iniciar la siguiente, el procesador organiza las fases como una línea de producción. Mientras una instrucción se encuentra en ejecución, otra puede estar siendo decodificada y una tercera puede ser recuperada desde memoria. Esta superposición no reduce necesariamente el tiempo que tarda una instrucción individual en atravesar todas las etapas, pero sí incrementa el número de instrucciones completadas por unidad de tiempo cuando el pipeline se mantiene lleno. Por ello, la segmentación debe distinguirse entre latencia y throughput. La latencia de una instrucción puede mantenerse o incluso aumentar ligeramente por el costo de registros intermedios y control de etapas; el throughput, en cambio, mejora cuando la tubería logra completar instrucciones de manera sostenida.

El modelo clásico de pipeline RISC de cinco etapas ofrece una estructura pedagógica especialmente útil para analizar esta dinámica. La etapa IF realiza la búsqueda de instrucción; ID decodifica la operación y lee registros; EX ejecuta la operación aritmética, lógica o calcula direcciones; MEM realiza el acceso a memoria cuando la instrucción lo requiere; y WB escribe el resultado en el banco de registros. Esta secuencia muestra que el procesador segmentado intenta distribuir el trabajo en unidades relativamente equilibradas para que cada etapa pueda operar durante un ciclo de reloj. Ponrani, Thanishtha y Swasthika (2026)

utilizan precisamente este modelo de cinco etapas como referencia para comparar procesadores RISC con control mínimo de hazards y diseños mejorados con forwarding, interlocks y manejo especial de dependencias, demostrando que el valor del pipeline depende de la calidad de sus mecanismos de coordinación.

La segmentación, sin embargo, no debe confundirse con paralelismo ilimitado. El pipeline permite que varias instrucciones estén activas al mismo tiempo, pero su eficiencia depende de que esas instrucciones puedan avanzar sin interferir entre sí. Cuando el flujo es regular, las etapas se mantienen ocupadas y el procesador se aproxima a la condición ideal de completar una instrucción por ciclo en un pipeline escalar. Pero cuando aparecen dependencias, saltos, conflictos de recursos o accesos irregulares, la tubería puede detenerse, vaciarse o introducir burbujas. En términos microarquitectónicos, estas interrupciones son conocidas como hazards, y representan el costo estructural de convertir una secuencia lógica en una ejecución solapada. La promesa del pipelining consiste en aumentar el flujo de instrucciones; su dificultad consiste en preservar corrección y eficiencia cuando las instrucciones ya no avanzan de manera aislada.

Los hazards estructurales aparecen cuando dos o más instrucciones requieren simultáneamente un mismo recurso físico. Por ejemplo, si una arquitectura utiliza una sola memoria para instrucciones y datos, una instrucción en IF puede necesitar acceder a memoria al mismo tiempo que otra instrucción en MEM intenta leer o escribir un dato. De manera similar, si varias instrucciones compiten por una única unidad funcional, un puerto de escritura o una ruta interna de datos, el procesador debe detener alguna etapa o duplicar recursos. La solución puede consistir en separar memorias de instrucciones y datos, aumentar puertos, replicar unidades funcionales o introducir planificación de acceso. No obstante, cada solución tiene costos de área, energía y complejidad, por lo que el diseño de un pipeline implica decidir qué conflictos serán evitados mediante hardware y cuáles serán gestionados mediante control.

Los hazards de datos se producen cuando una instrucción depende del resultado de otra que aún no ha completado su recorrido por el pipeline. El caso más común es la dependencia RAW, en la que una instrucción necesita leer un dato que una

instrucción anterior todavía no ha escrito. Si el procesador permite que la instrucción dependiente avance sin control, podría usar un valor obsoleto y producir un resultado incorrecto. Para evitarlo, el diseño puede insertar stalls, es decir, ciclos de espera, hasta que el dato esté disponible. Sin embargo, detener el pipeline reduce el throughput. Por ello, se introducen mecanismos de forwarding o bypassing, que permiten reenviar resultados desde etapas avanzadas del pipeline hacia etapas anteriores sin esperar a que se escriban formalmente en el banco de registros. Esta técnica disminuye esperas, pero exige rutas adicionales y lógica de detección de dependencias.

La dependencia load-use constituye uno de los casos más complejos dentro de los hazards de datos. Cuando una instrucción de carga obtiene un dato desde memoria, el valor puede estar disponible más tarde que un resultado producido directamente por la ALU. Si la instrucción siguiente necesita ese dato de inmediato, el forwarding puede no ser suficiente, porque el dato todavía no ha llegado al punto desde el cual puede reenviarse. En estos casos se requiere un interlock selectivo que detenga temporalmente la instrucción dependiente sin paralizar innecesariamente todo el sistema. Ponrani et al. (2026) muestran que un diseño mejorado con rutas de bypass extendidas y mecanismos selectivos de interlock puede reducir significativamente los ciclos de espera frente a un procesador base, lo que confirma que el rendimiento del pipeline no depende solo de dividir etapas, sino de resolver con precisión los casos límite de dependencia.

Los hazards de control están asociados con saltos, bifurcaciones y cambios en el flujo del programa. En un pipeline, la CPU continúa buscando instrucciones mientras todavía puede no conocer si una rama será tomada o no. Si el procesador espera hasta resolver el salto, desperdicia ciclos; si continúa por una dirección estimada y se equivoca, debe descartar las instrucciones incorrectas que ingresaron al pipeline. Este problema se vuelve más severo en pipelines profundos, porque una mala decisión puede implicar vaciar muchas etapas. En los procesadores superscalar, la dificultad aumenta porque se pueden haber buscado y emitido múltiples instrucciones especulativas por ciclo. Pizzol, Pilla y Navaux (s. f.) señalan que las dependencias de control reducen el tamaño efectivo de los bloques básicos e impiden alimentar adecuadamente las etapas de

ejecución, razón por la cual la predicción de saltos se vuelve fundamental para mantener alto IPC en arquitecturas superscalar.

El pipeline introduce, por tanto, una tensión entre regularidad ideal y comportamiento real del programa. Los programas contienen dependencias, saltos, llamadas a funciones, accesos a memoria, interrupciones y excepciones. La microarquitectura debe convertir ese flujo irregular en una ejecución lo más continua posible. Para ello, utiliza detección de hazards, forwarding, interlocks, predicción, registros intermedios, control de flush, señales de stall y, en arquitecturas más avanzadas, renombramiento de registros y planificación dinámica. La segmentación no es simplemente una técnica de aceleración, sino una disciplina de administración temporal de instrucciones. Cada etapa debe avanzar cuando puede hacerlo y detenerse cuando avanzar comprometería la corrección.

La diferencia entre procesadores segmentados y no segmentados también permite comprender la evolución del diseño microarquitectónico. En un procesador no segmentado, la simplicidad de control puede favorecer predictibilidad, menor área y menor complejidad, pero limita el flujo de instrucciones. En un procesador segmentado, el rendimiento potencial aumenta, aunque a costa de mayor lógica de control, registros interetapa y mecanismos de resolución. Jackson (2007), al estudiar microprocesadores seriales para FPGA, muestra que la simplificación extrema del procesador puede reducir uso de lógica, pero también incrementa el número de ciclos requeridos para procesar datos, lo que obliga a evaluar el rendimiento no solo por frecuencia, sino por relación entre lógica utilizada, ciclos consumidos y capacidad de ejecución.

Esta observación es relevante porque el pipelining no siempre es la solución óptima para todos los contextos. En procesadores de propósito general y alto rendimiento, la segmentación es indispensable para sostener throughput. En microcontroladores simples, sistemas embebidos ultraligeros o diseños FPGA con recursos limitados, una tubería compleja puede no justificar su costo. La arquitectura debe ajustarse al dominio de aplicación. Un sistema que prioriza baja latencia determinista, bajo consumo o mínima área puede preferir un diseño más simple; uno orientado a ejecución intensiva puede requerir segmentación

profunda, predicción agresiva y múltiples unidades funcionales. Por ello, la segmentación debe evaluarse como una estrategia dentro de un espacio de compromisos, no como un valor absoluto.

En FPGA, esta discusión adquiere matices particulares. El pipelining puede elevar la frecuencia máxima al reducir la lógica combinacional entre registros, pero también aumenta la cantidad de registros, rutas de control y complejidad de sincronización. Los procesadores softcore deben adaptarse a recursos como LUTs, flip-flops, bloques de memoria e interconexión programable. Jackson (2007) plantea que una arquitectura bit-serial puede ocupar muy pocos recursos lógicos y operar a una frecuencia relativamente alta, aunque requiera más ciclos, lo que abre una lectura alternativa del rendimiento basada en desempeño por unidad de lógica y no únicamente en instrucciones por segundo. Esta perspectiva resulta útil para comprender que la segmentación profunda, típica de procesadores comerciales, no es necesariamente deseable en todos los entornos de implementación.

En microprocesadores modernos, la segmentación también se relaciona con la frecuencia de reloj. Al dividir una operación larga en etapas más pequeñas, cada etapa puede completarse en un ciclo más corto, permitiendo aumentar la frecuencia. Sin embargo, profundizar el pipeline incrementa la penalización por fallos de predicción, la cantidad de registros intermedios, el consumo de control y la sensibilidad a dependencias. La búsqueda de altas frecuencias fue una fuerza importante en la evolución del procesador, pero sus beneficios se vieron limitados por disipación térmica y consumo energético. Nikolić et al. (2022) explican que la velocidad del microprocesador está condicionada por la frecuencia de reloj, pero esta no puede incrementarse indefinidamente porque el aumento de frecuencia eleva la disipación de potencia y el calentamiento, impulsando el tránsito hacia arquitecturas multicore y paralelismo.

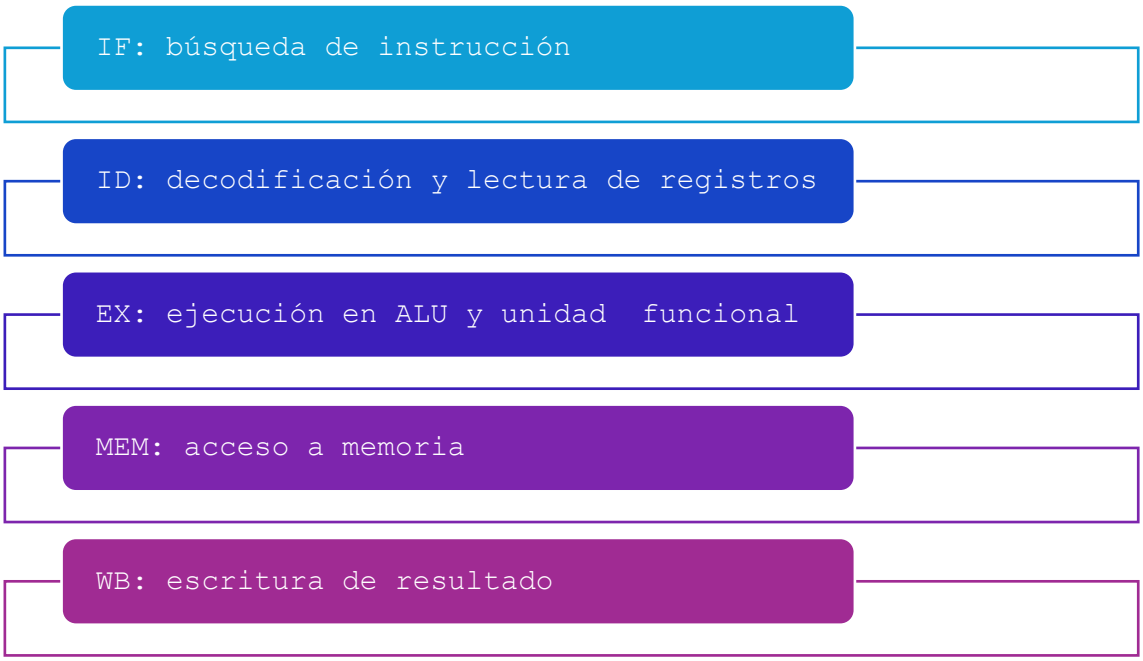
La segmentación instruccional debe entenderse, entonces, como una fase fundamental en la historia del paralelismo interno del procesador. Antes de llegar a superscalar, SMT, multicore o manycore, el pipeline introdujo la idea de que una CPU podía trabajar simultáneamente en varias instrucciones, aunque cada una estuviera en una fase distinta. Esta simultaneidad parcial abrió el camino a

formas más agresivas de paralelismo. Al mismo tiempo, reveló que el rendimiento no se obtiene únicamente agregando etapas, sino controlando dependencias. La microarquitectura moderna hereda esta lección: toda estrategia de paralelismo produce nuevas formas de coordinación, y toda coordinación tiene costos de energía, área, complejidad y verificabilidad.

Desde un punto de vista académico, el ciclo de instrucción y el pipeline son fundamentales porque permiten al estudiante comprender que la CPU no ejecuta programas de manera abstracta. Cada instrucción avanza por rutas físicas, atraviesa registros, activa multiplexores, depende de señales de control y puede interferir con otras instrucciones. La representación IF, ID, EX, MEM y WB es, por tanto, más que un esquema didáctico; es una herramienta para visualizar el tiempo interno de la máquina. Al observar dónde se producen hazards, stalls o forwarding, se comprende que la eficiencia computacional es una propiedad emergente de la interacción entre programa y microarquitectura. El procesador moderno no solo "hace cálculos"; organiza temporalmente el flujo de instrucciones para que la mayor cantidad posible de trabajo útil ocurra en paralelo sin romper la semántica del programa.

Para visualizar el funcionamiento de la segmentación instruccional, conviene representar el pipeline como una secuencia de etapas en la que cada fase cumple una función específica, pero también como una zona donde pueden aparecer interrupciones del flujo ideal. El valor de esta figura no reside solo en mostrar el orden IF, ID, EX, MEM y WB, sino en evidenciar que los hazards se producen precisamente cuando la superposición temporal de instrucciones encuentra dependencias, conflictos o incertidumbre de control. Esta representación puede elaborarse en Word mediante SmartArt de tipo "Proceso básico" o "Proceso en cheurones", agregando debajo tres etiquetas explicativas sobre hazards estructurales, de datos y de control.

Figura 3. ***Ciclo de instrucción segmentado y puntos críticos de hazard***



Nota. Elaboración propia a partir del modelo clásico de pipeline RISC de cinco etapas y de los aportes sobre segmentación, hazards, forwarding e interlocks desarrollados por Ponrani et al. (2026), Nikolić et al. (2022), Pizzol et al. (s. f.) y Jackson (2007).

La figura permite observar que el pipeline no es simplemente una línea secuencial de ejecución, sino una estructura temporal de coordinación. En condiciones ideales, cada etapa recibe una instrucción distinta en cada ciclo, de modo que el procesador mantiene ocupados sus recursos internos y completa resultados con mayor frecuencia. No obstante, los puntos críticos identificados muestran que el rendimiento efectivo depende de que el flujo se mantenga estable. Si una instrucción necesita un dato que todavía no se ha producido, si dos etapas compiten por el mismo recurso o si el procesador desconoce la dirección correcta de un salto, la tubería pierde continuidad. En consecuencia, el pipeline debe estar acompañado por mecanismos capaces de detectar, anticipar o corregir estas interrupciones.

La técnica de forwarding constituye una de las soluciones más importantes frente a los hazards de datos. Su lógica consiste en evitar que una instrucción dependiente espere hasta que el resultado de una instrucción anterior sea escrito formalmente en el banco de registros. En lugar de ello, el resultado se reenvía directamente desde una etapa intermedia, como EX/MEM o MEM/WB, hacia la

entrada de la ALU o hacia la etapa que lo requiera. Esta operación parece sencilla en términos funcionales, pero implica agregar multiplexores, comparadores de registros, rutas de bypass y lógica de selección. La ventaja es que se reducen stalls y se mantiene mayor continuidad en la tubería. La desventaja es que el datapath se vuelve más complejo, aumenta el consumo de área y se incrementa la carga de verificación, especialmente cuando se agregan dependencias especiales, registros de propósito particular o instrucciones con latencias variables.

Los interlocks, por su parte, actúan como mecanismos de protección cuando el forwarding no puede resolver una dependencia. Su función es detener selectivamente una etapa o insertar burbujas para evitar que una instrucción avance con datos incorrectos. En un diseño bien equilibrado, el interlock no debe ser una pausa general indiscriminada, sino una respuesta precisa ante una condición concreta de riesgo. Esta precisión es relevante porque los stalls innecesarios reducen el throughput y desperdician la ventaja del pipeline. Ponrani et al. (2026) reportan que un diseño avanzado de cinco etapas con forwarding extendido, interlock selectivo y manejo de registros especiales redujo significativamente el tiempo de resolución de hazards y los ciclos de espera frente a una versión base, lo cual confirma que el rendimiento depende de la fineza de los mecanismos de control y no únicamente de la existencia de segmentación.

El flush representa una estrategia distinta, asociada principalmente a hazards de control. Cuando el procesador continúa buscando instrucciones después de un salto y luego descubre que la dirección seguida era incorrecta, debe invalidar las instrucciones especulativas que ingresaron al pipeline. Este vaciamiento parcial o total de la tubería produce pérdida de ciclos, porque el trabajo realizado sobre la ruta equivocada no puede contribuir al resultado final del programa. En pipelines simples, la penalización puede ser moderada; en pipelines profundos, superscalar o con emisión múltiple, la penalización crece. La predicción de saltos intenta reducir la frecuencia de estos eventos, pero nunca los elimina por completo. Por ello, la arquitectura moderna combina predicción, recuperación rápida, especulación controlada y mecanismos de commit para preservar la semántica correcta.

La segmentación también requiere una reflexión sobre el balance entre etapas. En un pipeline ideal, todas las fases tienen duraciones similares, de modo que el ciclo de reloj puede ajustarse al retardo de la etapa más lenta sin desperdiciar excesivamente las demás. En la práctica, algunas fases tienden a ser más costosas, como el acceso a memoria, la ejecución de operaciones complejas o la decodificación de instrucciones variables. Si una etapa es mucho más lenta que las demás, se convierte en el límite del ciclo de reloj. Si se divide en subetapas, puede aumentar la frecuencia, pero también crece el número de registros intermedios y la penalización por control. Esta tensión muestra que el diseño del pipeline es una negociación entre profundidad, frecuencia, latencia, energía y complejidad.

La relación entre pipeline e ISA es especialmente importante. Las arquitecturas RISC fueron históricamente favorables a la segmentación porque sus instrucciones regulares, su longitud fija y su modelo load-store simplifican la decodificación y la organización de etapas. En cambio, las arquitecturas CISC clásicas, con instrucciones de longitud variable y modos de direccionamiento más complejos, presentaban mayores dificultades para una segmentación simple. Sin embargo, los procesadores CISC modernos han incorporado mecanismos internos que traducen instrucciones complejas en micro-operaciones más regulares, permitiendo aplicar técnicas similares a las usadas en microarquitecturas RISC. Esta evolución demuestra que la segmentación no depende únicamente de la filosofía externa de la ISA, sino de la manera en que la implementación interna organiza el flujo de ejecución.

En este punto aparece una distinción esencial entre arquitectura visible y microarquitectura. Desde el punto de vista del programador, el procesador ejecuta instrucciones definidas por la ISA; desde el punto de vista interno, esas instrucciones pueden atravesar múltiples etapas, ser transformadas, reordenadas, renombradas, especuladas o divididas en operaciones más pequeñas. El pipeline pertenece al plano microarquitectónico, porque expresa cómo el procesador implementa físicamente el comportamiento prometido por la ISA. Esta distinción permite comprender por qué dos procesadores compatibles con una misma ISA pueden tener rendimientos muy diferentes. Uno puede poseer un pipeline corto y simple; otro, un pipeline profundo, superscalar, con

predicción avanzada y ejecución fuera de orden. El contrato funcional puede ser el mismo, pero la organización temporal de la ejecución cambia radicalmente.

El pipelining también tiene efectos sobre la predictibilidad temporal. En sistemas de propósito general, la variabilidad introducida por cachés, pipelines y especulación puede ser aceptable si el rendimiento promedio mejora. En sistemas de control en tiempo real, en cambio, la variabilidad puede convertirse en un problema. Proctor y Shackleford (s. f.) analizan cómo características de microprocesadores generales, como cachés, pipelines y ejecución especulativa, introducen incertidumbre temporal que puede afectar aplicaciones de control, especialmente cuando las tareas requieren períodos de planificación muy reducidos. Esta observación permite problematizar la segmentación desde otra perspectiva: una técnica diseñada para mejorar rendimiento promedio puede dificultar el cálculo preciso del peor caso temporal.

La tensión entre rendimiento promedio y determinismo es clave en sistemas embebidos, robótica, control industrial y aplicaciones críticas. Un pipeline profundo puede mejorar throughput en cargas generales, pero si introduce variaciones por stalls, misses, predicción incorrecta o interferencias con tareas no críticas, puede afectar la regularidad temporal exigida por ciertos sistemas. Por ello, algunos procesadores embebidos prefieren pipelines más simples, menor especulación y comportamiento más predecible. La eficiencia, en estos casos, no se mide solo por instrucciones por ciclo, sino por capacidad de garantizar tiempos máximos, reducir jitter y responder a eventos externos dentro de ventanas estrictas. Así, el diseño del pipeline debe alinearse con el tipo de sistema y no solamente con el rendimiento agregado.

La segmentación se relaciona asimismo con el consumo energético. Cada etapa del pipeline requiere registros, señales de control y lógica adicional. A medida que el pipeline se profundiza, aumenta la cantidad de elementos que deben activarse por ciclo, y la especulación puede producir trabajo que luego será descartado. Si una predicción falla, el procesador habrá consumido energía en instrucciones que no formarán parte de la ejecución válida. Esta situación evidencia que el rendimiento obtenido mediante pipelining no es gratuito. La microarquitectura debe evaluar cuánto trabajo útil produce por unidad de energía y cuánta actividad

se desperdicia en stalls, flushes, caminos erróneos o mecanismos de control. En la era de restricciones térmicas y dispositivos móviles, esta dimensión es tan importante como la frecuencia de reloj.

El pipeline también actúa como fundamento de técnicas más avanzadas. Los procesadores superscalar amplían el concepto al permitir que varias instrucciones ingresen o avancen por etapas paralelas en un mismo ciclo. La ejecución fuera de orden complementa la segmentación al permitir que instrucciones listas adelanten a otras detenidas. El multithreading simultáneo intenta aprovechar burbujas del pipeline emitiendo instrucciones de otros hilos. Los procesadores multicore multiplican unidades de ejecución completas, desplazando el paralelismo desde el interior de un pipeline hacia varios núcleos. Jesshope y Luo (2000) plantean que la búsqueda de mayor IPC en microprocesadores RISC condujo a técnicas como superpipelining, emisión múltiple, predicción dinámica, renombramiento de registros y especulación, pero advierten que estas estrategias incrementan la complejidad y no siempre producen aumentos proporcionales al ancho de la tubería.

Esta advertencia es importante porque muestra que el pipeline tiene rendimientos decrecientes. Dividir más etapas, emitir más instrucciones o ampliar ventanas no garantiza mejoras lineales. Los programas poseen dependencias reales, las ramas limitan el flujo, la memoria introduce latencias y los recursos físicos consumen energía y área. Por ello, la microarquitectura moderna combina técnicas, pero también reconoce límites. El pipeline fue una innovación decisiva, pero sus beneficios dependen del contexto. Su eficiencia máxima se alcanza cuando el flujo de instrucciones es regular, las dependencias son manejables, la predicción es acertada, la memoria responde oportunamente y las unidades funcionales no permanecen ociosas.

En términos de diseño educativo, el estudio del pipeline permite formar una comprensión más profunda de la relación entre programación y hardware. Un estudiante que observa únicamente código fuente puede imaginar que las instrucciones se ejecutan como una secuencia simple. El análisis segmentado revela, en cambio, que el orden escrito no siempre coincide con el comportamiento temporal interno, y que la eficiencia depende de cómo las

instrucciones interactúan con la máquina. Bucles, saltos, dependencias de datos, acceso a memoria y llamadas a funciones no son solo estructuras del lenguaje, sino patrones que afectan la dinámica del pipeline. Esta comprensión es esencial para cursos de arquitectura, compiladores, sistemas embebidos, optimización y computación de alto rendimiento.

Desde el punto de vista de la ingeniería, la segmentación obliga a diseñar simultáneamente datapath y control. No basta con dividir la ejecución en etapas; es necesario definir qué registros separan cada etapa, qué señales se propagan, cómo se detectan dependencias, qué valores pueden adelantarse, qué instrucciones se invalidan, cómo se actualiza el contador de programa y cómo se preserva el estado arquitectónico ante excepciones. El pipeline es, por tanto, una estructura de control temporal. Su corrección depende de que cada instrucción produzca el mismo resultado que habría producido en una ejecución secuencial ideal, aunque internamente comparta el procesador con otras instrucciones en diferentes fases.

La complejidad de la segmentación también explica por qué la verificación de procesadores es una tarea crítica. Los hazards no resueltos pueden producir errores sutiles, difíciles de detectar mediante pruebas superficiales. Una dependencia mal identificada, una señal de forwarding incorrecta, un flush incompleto o una excepción gestionada fuera de orden pueden comprometer la coherencia del sistema. En FPGA y diseños académicos, la simulación y la síntesis permiten observar estos problemas con claridad, pero en procesadores comerciales la escala es mucho mayor. Por ello, la segmentación instruccional no es solo un tema de rendimiento, sino también de confiabilidad del diseño.

Puede afirmarse que el ciclo de instrucción y el pipelining constituyen la base del paralelismo interno del microprocesador moderno. La segmentación transforma la ejecución secuencial en un flujo solapado, incrementa throughput y prepara el camino para técnicas más agresivas como superscalar, predicción y ejecución fuera de orden. No obstante, también introduce hazards, complejidad de control, consumo energético, variabilidad temporal y necesidad de verificación rigurosa. La arquitectura moderna hereda esta tensión: para ejecutar más rápido debe superponer más trabajo; para superponer más trabajo debe anticipar, detectar y

corregir más riesgos. El pipeline, en consecuencia, no es solo una técnica de aceleración, sino una expresión concreta de la forma en que el hardware administra el tiempo interno de la computación.

Conjuntos de instrucciones RISC vs. CISC

La comparación entre RISC y CISC constituye uno de los debates más persistentes en la historia de la arquitectura de computadores, pero su interpretación contemporánea exige superar lecturas simplificadas. Tradicionalmente, RISC se ha asociado con instrucciones simples, longitud fija, pocos modos de direccionamiento, modelo load-store y facilidad para implementar pipelines regulares. CISC, por su parte, se ha vinculado con instrucciones más complejas, longitud variable, múltiples modos de direccionamiento, operaciones capaces de realizar varias acciones de bajo nivel y mayor densidad de código. Esta oposición tuvo gran valor histórico para explicar dos filosofías de diseño, pero los microprocesadores modernos han difuminado sus fronteras. Hoy, muchas arquitecturas CISC traducen internamente sus instrucciones complejas en micro-operaciones más regulares, mientras que los procesadores RISC incorporan mecanismos internos altamente sofisticados, como predicción dinámica, ejecución fuera de orden, renombramiento de registros, cachés multinivel, unidades vectoriales y control energético avanzado.

La distinción entre ISA y microarquitectura es fundamental para comprender este debate. La ISA define el repertorio de instrucciones visibles para el software, los tipos de datos, registros, modos de direccionamiento y convenciones de ejecución. La microarquitectura, en cambio, determina cómo esa ISA se implementa físicamente mediante pipelines, unidades funcionales, cachés, predictores, decodificadores, buses internos y políticas de energía. Por ello, dos procesadores con una misma ISA pueden tener rendimientos muy distintos, y dos procesadores con ISAs diferentes pueden converger en soluciones internas parecidas. Sultanpuri (2025) explica que la ISA funciona como la interfaz mediante la cual el software controla el hardware, pero también subraya que las ISAs evolucionan mediante extensiones, ampliación de datos, instrucciones especializadas y adaptación a necesidades específicas, como ocurre con RISC-V y sus módulos orientados a dominios particulares.

Desde la perspectiva RISC, la simplicidad de las instrucciones favorece la regularidad del pipeline. Si las instrucciones tienen longitud fija, formatos uniformes y operaciones relativamente simples, el procesador puede buscar, decodificar y ejecutar con menor complejidad estructural. Esta regularidad contribuye a facilitar la segmentación, la predicción de tiempos, la implementación de hardware más compacto y la optimización por parte de compiladores. En sistemas embebidos y de bajo consumo, estas características resultan especialmente valiosas porque reducen complejidad de control, área de silicio y consumo dinámico. Rosdiyanto, Zuhro y Nisfuwadi (2025) sostienen que las arquitecturas RISC, especialmente ARM Cortex y RISC-V, resultan adecuadas para aplicaciones de bajo consumo, IoT y sistemas en tiempo real debido a su modularidad, eficiencia energética, predictibilidad y adaptación a toolchains ligeros.

Sin embargo, la simplicidad de las instrucciones RISC no significa ausencia de complejidad. En muchos procesadores modernos, la ejecución eficiente de una ISA aparentemente simple requiere cachés sofisticadas, predicción de saltos, unidades de ejecución múltiples, soporte SIMD, mecanismos de coherencia, administración energética y extensiones especializadas. La complejidad se desplaza desde la instrucción individual hacia la organización microarquitectónica y hacia el ecosistema de compilación. En este sentido, RISC no debe entenderse como hardware elemental, sino como una filosofía de regularidad externa que puede convivir con implementaciones internas muy avanzadas. La ventaja no reside en que el procesador sea necesariamente simple en todos sus niveles, sino en que el contrato visible para el software sea suficientemente regular para facilitar segmentación, optimización y escalabilidad.

En CISC, la lógica histórica fue distinta. Las instrucciones complejas buscaban reducir la distancia entre lenguajes de alto nivel y código máquina, mejorar densidad de código y realizar operaciones más ricas con menos instrucciones. Este enfoque resultaba atractivo cuando la memoria era costosa y la programación en bajo nivel exigía representar operaciones frecuentes de manera compacta. La densidad de código sigue siendo una ventaja relevante en ciertos entornos, porque programas más compactos pueden reducir presión sobre

memoria de instrucciones y caché. No obstante, la complejidad de decodificación y la variabilidad de instrucciones dificultan una ejecución segmentada directa. Por ello, los procesadores x86 contemporáneos suelen transformar instrucciones CISC en micro-operaciones internas más simples, creando una especie de núcleo de ejecución más regular bajo una ISA externa compleja.

Este fenómeno de traducción interna muestra que el debate RISC/CISC ya no puede formularse como una oposición pura entre simplicidad y complejidad. En la práctica, el procesador x86 moderno conserva compatibilidad con décadas de software, pero implementa técnicas internas que recuerdan principios RISC para sostener alto rendimiento. A su vez, procesadores ARM o RISC-V pueden incorporar extensiones vectoriales, unidades especializadas y jerarquías complejas que los alejan de una idea ingenua de reducción. Dominguez Perez (2016), al revisar el trabajo de Blem, Menon y Sankaralingam sobre ARM y x86 contemporáneos, resalta que las comparaciones modernas deben considerar rendimiento y energía desde mediciones de plataforma, porque las diferencias no dependen exclusivamente de la ISA, sino también de microarquitectura, memoria, proceso tecnológico, compilador y cargas de trabajo.

La comparación entre ARM y x86-64 en plataformas recientes confirma esta necesidad de una lectura sistémica. ARM suele asociarse con eficiencia energética, integración SoC, diseño heterogéneo y uso extendido en móviles, portátiles y sistemas embebidos. x86-64, en cambio, conserva una posición dominante en computación personal, servidores, estaciones de trabajo y ecosistemas de alto desempeño con amplia compatibilidad. Sin embargo, estas asociaciones son insuficientes si se usan como explicaciones absolutas. Un procesador ARM de alto rendimiento puede incluir núcleos complejos, memoria unificada, aceleradores, predicción avanzada y unidades vectoriales; un procesador x86-64 puede aplicar políticas agresivas de ahorro energético y micro-operaciones optimizadas. Özyılmaz (2026) advierte que las diferencias observadas entre una plataforma Apple M3 y una AMD Ryzen 7 3750H deben leerse como resultados de sistemas completos, donde ISA, memoria, organización de núcleos, integración y gestión energética interactúan con los benchmarks utilizados.

La energía ocupa un lugar decisivo en esta comparación contemporánea. En etapas anteriores, el debate RISC/CISC se centró con frecuencia en rendimiento, densidad de código y complejidad de instrucción. Hoy, la eficiencia energética se ha convertido en un criterio igualmente importante, especialmente en portátiles, dispositivos móviles, sensores, sistemas embebidos, edge computing y centros de datos. Una arquitectura que obtiene buen tiempo de ejecución pero consume energía excesiva puede ser menos conveniente que otra con rendimiento moderado y mejor energía por tarea. Esta perspectiva obliga a evaluar métricas como energía por solución, rendimiento por watt y consumo bajo cargas reales. Özyılmaz (2026) muestra precisamente que una plataforma puede no ser siempre la más rápida en todos los benchmarks, pero sí ofrecer diferencias sustantivas en energía consumida por carga completada, lo que exige separar rendimiento bruto de eficiencia energética.

La eficiencia energética no depende únicamente de la ISA. Intervienen la tecnología de fabricación, la organización de núcleos, el escalado dinámico de voltaje y frecuencia, la gestión de estados de bajo consumo, la integración de memoria, la política térmica y la capacidad del sistema operativo para asignar tareas a núcleos adecuados. Las arquitecturas heterogéneas, frecuentes en diseños ARM modernos, combinan núcleos de alto rendimiento con núcleos de alta eficiencia, permitiendo ejecutar cargas ligeras con bajo consumo y cargas intensivas con mayor capacidad. Esta estrategia desplaza el análisis desde el repertorio de instrucciones hacia la planificación energética del sistema. La ISA puede favorecer ciertas optimizaciones, pero el rendimiento por watt surge de la coordinación entre hardware, firmware, sistema operativo y carga de trabajo.

RISC-V introduce una tercera dimensión en el debate porque no se limita a reproducir la oposición RISC/CISC tradicional, sino que agrega apertura, modularidad y extensibilidad. Su diseño permite partir de un conjunto base reducido y añadir extensiones según necesidades concretas, como multiplicación, punto flotante, instrucciones comprimidas, operaciones atómicas, vectores o packed-SIMD. Esta modularidad resulta atractiva para investigación, educación, industria emergente y diseño de procesadores específicos de dominio, porque evita cargar todos los sistemas con instrucciones innecesarias. Sultanpuri (2025) destaca que RISC-V surge como alternativa abierta frente a ISAs propietarias, y

que su extensión P permite incorporar packed-SIMD para aplicaciones DSP, reduciendo ciclos e instrucciones retiradas en ciertos kernels representativos.

El carácter extensible de RISC-V permite retomar una idea clave: la eficiencia puede requerir especialización controlada. Una ISA mínima puede ser suficiente para ejecutar programas generales, pero no necesariamente óptima para dominios como señales, visión artificial, criptografía o aprendizaje automático. Las extensiones permiten agregar instrucciones que aceleran patrones frecuentes sin abandonar la estructura general de la arquitectura. Este enfoque conecta la filosofía RISC con el diseño de procesadores específicos de aplicación, donde el objetivo no es tener la instrucción más compleja posible, sino la instrucción adecuada para reducir ciclos, consumo o movimiento de datos en una carga concreta. La extensión P, orientada a SIMD empacado, ejemplifica esta lógica porque permite operar sobre varios elementos de datos dentro de un mismo registro, aprovechando paralelismo de datos en aplicaciones DSP.

En sistemas embebidos, la elección entre RISC, CISC o una variante abierta como RISC-V debe considerar restricciones de tiempo real, consumo, memoria, costo, certificación y ecosistema. Un microcontrolador ARM puede ser preferible por madurez de herramientas, documentación y soporte industrial. RISC-V puede ser atractivo por apertura, personalización y ausencia de licencias restrictivas. x86 embebido puede ser conveniente cuando se requiere compatibilidad con software legado, sistemas operativos complejos o integración con entornos industriales existentes. Rosdiyanto et al. (2025) sostienen que la selección de arquitectura en sistemas embebidos modernos debe atender eficiencia energética, complejidad de diseño, respuesta a interrupciones, uso de memoria y soporte de herramientas, en lugar de asumir superioridad universal de un paradigma.

La densidad de código también merece una lectura crítica. CISC puede representar ciertas operaciones con menos instrucciones, lo que reduce tamaño binario y potencialmente presión sobre memoria de instrucciones. RISC, al usar instrucciones simples, puede requerir más instrucciones para expresar la misma tarea, aunque cada una sea más fácil de decodificar y segmentar. Las instrucciones comprimidas, presentes en ciertas ISAs RISC como RISC-V C extension, intentan resolver esta tensión mediante formatos más compactos para

operaciones frecuentes. Así, incluso dentro de RISC aparece una preocupación por densidad de código que históricamente se atribuía a CISC. La evolución muestra que las arquitecturas no se mantienen puras, sino que incorporan soluciones pragmáticas para equilibrar regularidad, compacidad y rendimiento.

El papel del compilador es igualmente decisivo. Una ISA regular puede facilitar la generación de código eficiente, pero el rendimiento final depende de la capacidad del compilador para organizar instrucciones, asignar registros, explotar paralelismo, vectorizar bucles, reducir saltos y aprovechar extensiones específicas. En arquitecturas CISC, el compilador puede elegir subconjuntos de instrucciones que se comportan mejor en microarquitecturas modernas, evitando instrucciones complejas poco eficientes. En arquitecturas RISC, puede desplegar secuencias más largas pero más predecibles. La frontera entre ISA y compilador es, por tanto, un espacio de co-diseño. Un conjunto de instrucciones no es eficiente por sí mismo si las herramientas no logran explotarlo adecuadamente.

Esta relación se vuelve especialmente clara en procesadores específicos de aplicación. Lee, Choi y Dutt (2003) plantean que la síntesis de instrucciones específicas puede mejorar la eficiencia energética de procesadores ASIP al considerar codificación de instrucciones, conteo dinámico y producto energía-retardo, lo que permite generar instrucciones complejas orientadas a reducir ciclos o consumo en dominios particulares. Esta propuesta problematiza la frontera clásica entre RISC y CISC: en ciertos contextos, una instrucción más compleja puede ser energéticamente conveniente si reemplaza una secuencia larga de instrucciones simples y reduce actividad dinámica. Lo importante no es la complejidad formal de la instrucción, sino su contribución real al rendimiento y a la energía bajo una carga concreta.

La discusión RISC/CISC también se vincula con la enseñanza de arquitectura. Presentar ambos paradigmas como categorías rígidas puede ayudar en una primera aproximación, pero limita la comprensión de los procesadores actuales. En cursos universitarios de organización de computadores y arquitectura, la comparación debe evolucionar hacia una visión donde ISA, microarquitectura, compilador, memoria, energía y aplicación formen una unidad de análisis. El syllabus revisado del programa de Computer Science and Engineering incluye

asignaturas como Computer Organisation, Microprocessors & Microcontrollers y Computer Architecture, lo que muestra que estos conceptos se estructuran curricularmente como fundamentos progresivos para comprender desde la lógica digital hasta el diseño de procesadores y sistemas complejos (JIS College of Engineering, 2015).

En conclusión parcial, RISC y CISC deben entenderse hoy como tradiciones de diseño que influyeron profundamente en la arquitectura de procesadores, pero no como compartimentos cerrados. RISC aportó regularidad, facilidad de segmentación y eficiencia en dominios de bajo consumo; CISC aportó densidad de código, compatibilidad y riqueza semántica; RISC-V agregó apertura, modularidad y extensibilidad; los procesadores modernos, por su parte, integran técnicas internas que combinan elementos de varias tradiciones. La pregunta relevante ya no es qué paradigma es universalmente superior, sino qué combinación de ISA, microarquitectura, compilador, memoria, energía y ecosistema resulta más adecuada para una carga determinada.

La comparación contemporánea entre RISC y CISC debe incorporar, además, el papel de las micro-operaciones internas. En los procesadores x86 modernos, muchas instrucciones visibles para el software no son ejecutadas directamente como operaciones monolíticas, sino traducidas por el front-end del procesador hacia micro-operaciones más simples que luego circulan por unidades de ejecución internas. Esta estrategia permite conservar compatibilidad binaria con software histórico, mientras se aprovechan técnicas microarquitectónicas asociadas a pipelines más regulares, emisión múltiple, renombramiento de registros y ejecución fuera de orden. En consecuencia, la ISA externa conserva una apariencia CISC, pero la ejecución interna adopta rasgos de regularidad que históricamente se asociaron con RISC. Dominguez Perez (2016), al sintetizar la discusión sobre ARM y x86 contemporáneos, destaca precisamente que las instrucciones CISC modernas pueden ser divididas en micro-operaciones de tipo RISC, mientras los compiladores tienden a seleccionar instrucciones que favorecen la eficiencia microarquitectónica real y no solo la expresividad formal de la ISA.

Esta condición obliga a distinguir entre complejidad visible y complejidad oculta. En CISC, parte de la complejidad se mantiene en el decodificador, en la traducción interna y en los mecanismos de compatibilidad. En RISC, la complejidad puede trasladarse hacia el compilador, hacia secuencias más largas de instrucciones o hacia extensiones especializadas. Ninguno de los dos paradigmas elimina la complejidad; la redistribuye. Por ello, una comparación madura debe preguntar dónde se ubica la complejidad, quién la administra y qué costo tiene en rendimiento, energía, área, verificación y soporte de software. En sistemas de escritorio o servidores, puede ser razonable pagar una complejidad considerable para conservar compatibilidad y alto rendimiento. En sistemas embebidos de baja potencia, en cambio, la simplicidad de decodificación y la previsibilidad temporal pueden resultar más valiosas que una ISA expresiva.

La longitud de instrucción es otro punto donde las diferencias históricas se han reinterpretado. Las instrucciones RISC de longitud fija simplifican la búsqueda y decodificación, porque el procesador puede identificar límites de instrucción de manera regular y alimentar el pipeline con menos ambigüedad. Las instrucciones CISC de longitud variable mejoran densidad de código, pero exigen una etapa de decodificación más compleja. En procesadores modernos, esta tensión se mitiga con cachés de micro-operaciones, decodificadores múltiples, predicción de longitud, predecodificación y mecanismos que amortiguan el costo de traducción. Sin embargo, tales mecanismos consumen área y energía. Esta situación evidencia que una ganancia en compacidad puede requerir inversión microarquitectónica para no perjudicar el flujo de instrucciones.

El modelo load-store, característico de muchas arquitecturas RISC, también favorece la claridad del pipeline. En este enfoque, las operaciones aritméticas y lógicas se realizan sobre registros, mientras que el acceso a memoria queda reservado a instrucciones específicas de carga y almacenamiento. Esta separación simplifica la ejecución, facilita la detección de dependencias y permite organizar etapas de memoria de manera más explícita. En CISC, algunas instrucciones pueden operar directamente con operandos en memoria, lo que reduce número de instrucciones, pero aumenta la complejidad de ejecución y de gestión de latencias. No obstante, los procesadores CISC contemporáneos pueden transformar internamente esas instrucciones en secuencias de micro-

operaciones load-store, lo que confirma nuevamente la convergencia práctica entre paradigmas.

La relación entre RISC/CISC y memoria es especialmente importante en aplicaciones actuales. Una ISA con instrucciones compactas puede reducir presión sobre la caché de instrucciones; una ISA regular puede facilitar prefetching, decodificación y planificación. Pero el rendimiento real depende de la jerarquía de memoria, de la tasa de fallos de caché, de la localidad de los datos, del ancho de banda y del patrón de acceso del programa. Özyılmaz (2026) muestra que las diferencias entre ARM y x86-64 no pueden atribuirse solo al repertorio de instrucciones, ya que los resultados de benchmarks dependen también del comportamiento de memoria, la organización del sistema, la energía medida y el tipo de carga, como ocurre al comparar un benchmark recursivo dominado por control frente a una multiplicación de matrices sensible a memoria y aritmética.

La comparación entre arquitecturas también se reconfigura cuando se consideran cargas de trabajo específicas. Una arquitectura puede destacar en tareas con muchas ramas y llamadas a funciones, mientras otra puede ser más eficiente en operaciones matriciales, procesamiento vectorial o cargas con alta localidad. Las diferencias en registros disponibles, convenciones de llamada, predictor de saltos, cachés, unidades vectoriales, política de frecuencia y sistema operativo pueden modificar los resultados. Esto impide formular una conclusión universal del tipo “RISC es más rápido” o “CISC es más potente”. La pregunta correcta debe formularse de manera contextual: qué arquitectura, en qué implementación, bajo qué carga, con qué compilador, en qué entorno energético y con qué restricciones de sistema.

En sistemas embebidos modernos, el debate se orienta cada vez más hacia eficiencia, latencia y sostenibilidad. La expansión de IoT, edge computing, sensores inteligentes y dispositivos alimentados por batería ha hecho que el consumo sea un criterio tan importante como el rendimiento. En este contexto, RISC-V y ARM han ganado relevancia por su capacidad de operar con bajo consumo, modularidad y adaptación a tareas específicas. Rosdiyanto et al. (2025) argumentan que RISC-V ofrece ventajas de flexibilidad y personalización en

sistemas embebidos modernos, mientras ARM mantiene un ecosistema maduro y CISC conserva pertinencia cuando se requiere compatibilidad con aplicaciones tradicionales o entornos industriales consolidados.

RISC-V merece atención particular porque transforma la discusión desde la oposición RISC/CISC hacia la gobernanza de la arquitectura. Al ser una ISA abierta y modular, permite que universidades, empresas emergentes, investigadores y fabricantes diseñen extensiones sin depender de modelos cerrados de licenciamiento. Esta apertura favorece experimentación en educación, investigación y diseño específico de dominio. No obstante, también exige ecosistemas robustos de compiladores, depuradores, simuladores, verificación, documentación y compatibilidad. La apertura por sí sola no garantiza rendimiento ni madurez industrial; su valor se materializa cuando se acompaña de herramientas, implementaciones confiables y comunidades técnicas activas. Sultanpuri (2025) evidencia este punto al mostrar que la implementación de una extensión P en un procesador RISC-V requiere no solo diseñar una ALU SIMD, sino también modificar toolchains, integrar rutas de control y validar kernels mediante métricas de ciclos e instrucciones retiradas.

Las extensiones SIMD representan un caso claro de convergencia entre simplicidad ISA y especialización. En lugar de ejecutar una operación sobre un solo dato, SIMD permite operar sobre varios elementos empaquetados dentro de un registro. Esta estrategia resulta especialmente útil para DSP, audio, imagen, visión artificial, filtros, matrices y aprendizaje automático. La incorporación de SIMD en arquitecturas RISC muestra que la simplicidad base no excluye instrucciones especializadas. Más bien, la modularidad permite agregar capacidades donde producen impacto. Esto obliga a matizar la idea de que RISC siempre se basa en instrucciones elementales. En las arquitecturas actuales, una instrucción puede ser simple en formato y decodificación, pero potente en efecto sobre múltiples datos.

La comparación RISC/CISC también debe considerar el costo de verificación. Una ISA más compleja, con múltiples modos de direccionamiento y comportamientos especiales, puede incrementar el espacio de casos que deben probarse. Una ISA más regular puede facilitar verificación, pero las extensiones

y microarquitecturas avanzadas vuelven a ampliar la complejidad. En procesadores modernos, la verificación no se limita a comprobar que cada instrucción funcione, sino que debe garantizar que la instrucción interactúe correctamente con pipeline, predicción, excepciones, interrupciones, memoria, caché, especulación y modos privilegiados. Así, la complejidad de la ISA y la complejidad microarquitectónica se combinan en un problema de confiabilidad.

El papel de la compatibilidad histórica no puede subestimarse. x86 mantiene enorme relevancia no solo por su rendimiento, sino por décadas de software, sistemas operativos, compiladores, bibliotecas y entornos empresariales construidos sobre su compatibilidad. Esta continuidad genera una ventaja difícil de reemplazar, incluso cuando otras arquitecturas ofrecen mayor eficiencia energética en ciertos dominios. CISC, por tanto, no sobrevive únicamente por rasgos técnicos internos, sino por su ecosistema. Del mismo modo, ARM no domina móviles solo por ser RISC, sino por su integración industrial, licenciamiento, eficiencia de diseños SoC y adaptación a dispositivos de bajo consumo. RISC-V no crecerá únicamente por ser abierto, sino por su capacidad de construir un ecosistema equivalente en herramientas, verificación y soporte.

En el ámbito educativo, esta discusión permite enseñar una lección conceptual importante: las categorías RISC y CISC son útiles si se presentan como tradiciones históricas, pero pueden ser engañosas si se usan como etiquetas absolutas. El estudiante debe comprender que el rendimiento de un procesador es resultado de una cadena de decisiones que incluyen ISA, microarquitectura, memoria, compilador, sistema operativo, energía, tecnología de fabricación y carga de trabajo. Un curso de arquitectura no debería limitarse a comparar listas de características, sino mostrar cómo cada filosofía de diseño resuelve ciertos problemas y desplaza otros. El syllabus de ingeniería en computación revisado ubica Computer Organisation, Microprocessors & Microcontrollers y Computer Architecture en una secuencia curricular progresiva, lo que confirma la necesidad de conectar fundamentos digitales, diseño de procesadores e interpretación de arquitecturas modernas (JIS College of Engineering, 2015).

Antes de presentar la tabla comparativa, conviene precisar que su propósito no es declarar la superioridad de una arquitectura, sino organizar criterios de

análisis para comprender sus diferencias y convergencias. En la actualidad, RISC, CISC y las arquitecturas híbridas modernas deben evaluarse desde una perspectiva de sistema completo, considerando no solo el repertorio de instrucciones, sino también pipeline, energía, compatibilidad, extensibilidad, memoria, toolchain y dominio de aplicación.

Tabla 6

Comparación contemporánea entre RISC, CISC y arquitecturas híbridas modernas

Criterio de análisis	RISC	CISC	Arquitecturas híbridas modernas
Filosofía de diseño	Prioriza instrucciones simples, regulares y fáciles de segmentar	Prioriza instrucciones más expresivas y densidad de código	Combina compatibilidad externa, traducción interna, extensiones y optimización microarquitectónica
Formato de instrucciones	Generalmente longitud fija o formatos regulares	Longitud variable y formatos más diversos	Puede mantener ISA externa compleja y ejecución interna mediante micro-operaciones regulares
Modelo de memoria	Predomina el enfoque load-store, con operaciones aritméticas sobre registros	Puede permitir operandos directos en memoria dentro de instrucciones complejas	Tiende a transformar operaciones complejas en secuencias internas más simples y gestionables
Pipeline y decodificación	Facilita segmentación por regularidad de instrucciones	Requiere decodificación más compleja	Usa decodificadores avanzados, cachés de micro-operaciones, predicción y front-end sofisticado
Densidad de código	Puede requerir más instrucciones para ciertas tareas, aunque existen formatos comprimidos	Suele ofrecer mejor compacidad por instrucción	Combina instrucciones compactas, extensiones y optimizaciones del compilador
Eficiencia energética	Alta pertinencia en móviles, IoT, embebidos y edge, aunque depende de la implementación	Puede ser menos eficiente en decodificación compleja, pero conserva ventajas por compatibilidad y optimizaciones	La eficiencia depende de tecnología, SoC, gestión energética, heterogeneidad y carga de trabajo
Compatibilidad histórica	Depende del ecosistema específico, con ARM muy consolidado y RISC-V en expansión	Muy fuerte en x86 por legado de software y sistemas empresariales	Busca preservar compatibilidad mientras incorpora técnicas internas modernas
Extensibilidad	RISC-V destaca por modularidad y extensiones abiertas; ARM incorpora extensiones propietarias	Extensibilidad más condicionada por compatibilidad y complejidad heredada	Integra SIMD, vectorización, aceleradores, micro-operaciones y unidades especializadas

Aplicaciones predominantes	Embebidos, móviles, IoT, edge, sistemas de bajo consumo, investigación abierta	Escritorio, servidores, estaciones de trabajo, sistemas industriales compatibles	Portátiles, servidores, IA, HPC, SoC heterogéneos y plataformas con aceleradores
Limitación principal	Puede requerir mayor número de instrucciones o depender fuertemente del compilador y extensiones	Complejidad de decodificación, consumo y carga de compatibilidad	Mayor complejidad de verificación, energía de control y dependencia del ecosistema completo

Nota. *Elaboración propia a partir de Özyılmaz (2026), Rosdiyanto et al. (2025), Dominguez Perez (2016), Sultanpuri (2025), Lee et al. (2003) y JIS College of Engineering (2015).*

La tabla evidencia que RISC y CISC no deben interpretarse como polos fijos, sino como tradiciones que han sido reinterpretadas por la evolución tecnológica. RISC conserva ventajas asociadas a regularidad, segmentación y eficiencia en sistemas de bajo consumo; CISC mantiene valor por compatibilidad, densidad de código y ecosistemas consolidados; las arquitecturas híbridas modernas integran elementos de ambos enfoques mediante traducción interna, extensiones vectoriales, aceleradores y control energético. De esta manera, el criterio de selección arquitectónica se desplaza desde la pureza conceptual hacia la adecuación funcional: una arquitectura es pertinente cuando responde a las restricciones de la aplicación, del sistema y del entorno de desarrollo.

La comparación también permite comprender que la ISA no es un destino tecnológico cerrado, sino una interfaz evolutiva. Las instrucciones pueden ampliarse, comprimirse, vectorizarse o especializarse; los procesadores pueden reinterpretarlas mediante micro-operaciones; los compiladores pueden seleccionar subconjuntos más eficientes; y las plataformas pueden compensar limitaciones mediante memoria, predicción, energía o aceleradores. En consecuencia, el rendimiento de una ISA no existe de manera abstracta. Existe como relación entre una especificación, una implementación, una carga de trabajo y un ecosistema.

Desde la perspectiva del libro, este subtema cumple una función articuladora. El ciclo de instrucción y el pipeline explican cómo se ejecutan instrucciones; la comparación RISC/CISC explica cómo se define el lenguaje visible de esas

instrucciones; la unidad de control microprogramada, desarrollada a continuación, permitirá comprender cómo ese lenguaje se traduce en señales internas de control. La arquitectura del microprocesador moderno se ubica precisamente en esa relación: un conjunto de instrucciones define posibilidades, una microarquitectura las implementa y un sistema completo determina si esa implementación resulta eficiente, sostenible y adecuada para los problemas contemporáneos.

Unidades de control microprogramadas

La unidad de control constituye el componente que organiza la actividad interna del procesador, porque transforma una instrucción codificada en una secuencia de señales capaces de activar registros, seleccionar rutas de datos, ordenar operaciones de la ALU, controlar accesos a memoria y coordinar la escritura de resultados. Sin ella, el datapath sería una colección de recursos incapaz de operar de manera coherente. La unidad de control define cuándo se habilita un registro, qué operación realiza la ALU, qué fuente alimenta un multiplexor, cuándo se lee o escribe memoria y cómo se actualiza el contador de programa. Por ello, aunque el procesador suele explicarse desde la ALU, los registros o el pipeline, la unidad de control es la estructura que convierte la arquitectura en una máquina secuenciada. En términos funcionales, representa el nivel donde la instrucción deja de ser un código binario y se convierte en comportamiento físico organizado.

Existen dos grandes enfoques para implementar la unidad de control: el control cableado y el control microprogramado. En el control cableado, las señales se generan mediante circuitos combinacionales y secuenciales diseñados específicamente para cada instrucción y estado de ejecución. Este enfoque puede ser rápido y eficiente cuando el repertorio de instrucciones es simple, pero se vuelve más complejo conforme crecen los modos de direccionamiento, las excepciones, las dependencias y los comportamientos especiales. En el control microprogramado, en cambio, las señales de control se almacenan como microinstrucciones dentro de una memoria de control. Cada instrucción visible para el programador puede ser implementada como una secuencia de microoperaciones, lo cual introduce un nivel intermedio entre la ISA y el hardware. Heinikoski (2023) recuerda que la microprogramación fue propuesta

por Maurice Wilkes en 1951 para simplificar la circuitería de los sistemas computacionales, al permitir que las instrucciones de máquina se implementaran mediante microprogramas en lugar de circuitos rígidos altamente complejos.

La microprogramación permite comprender con especial claridad la diferencia entre instrucción arquitectónica y microoperación. Una instrucción como sumar, cargar, almacenar o saltar puede parecer indivisible desde el punto de vista del lenguaje máquina, pero internamente requiere una serie de acciones elementales: colocar una dirección en el registro correspondiente, activar una lectura de memoria, transferir un dato hacia un registro interno, seleccionar operandos para la ALU, ejecutar una operación, actualizar banderas y escribir el resultado. Cada una de estas acciones puede ser descrita mediante señales de control. En una unidad microprogramada, dichas señales se agrupan en microinstrucciones que se ejecutan secuencial o condicionalmente, formando un microprograma. De esta manera, la microprogramación revela que la ejecución de instrucciones es una construcción jerárquica: el programa se compone de instrucciones, las instrucciones se componen de microoperaciones y las microoperaciones se materializan en señales eléctricas.

Esta estructura tiene un valor epistemológico importante para la arquitectura computacional, porque muestra que el hardware no es simplemente una base física opaca. Entre el software visible y los circuitos existe una capa de traducción organizada, capaz de ser diseñada, modificada, simulada y enseñada. En una máquina microprogramada, el comportamiento del procesador puede cambiarse modificando la memoria de control, siempre que el datapath disponga de los recursos necesarios. Esta flexibilidad resultó históricamente valiosa para implementar repertorios de instrucciones complejos, corregir errores de diseño, mantener compatibilidad y adaptar familias de procesadores sin rediseñar completamente la lógica de control. La microprogramación, por tanto, no es solo una técnica de implementación, sino una forma de separar el problema de qué debe hacer la máquina del problema de cómo debe activar físicamente sus recursos.

En la enseñanza de arquitectura de computadores, las máquinas microprogramadas poseen un valor formativo particular. Permiten que el

estudiante observe cómo una instrucción se descompone en pasos internos y cómo cada paso afecta registros, buses, memoria y ALU. Heinikoski (2023) desarrolló un simulador web de una máquina microprogramada con el objetivo de que los usuarios escriban microprogramas, los ejecuten y observen cómo reaccionan los componentes del computador en un entorno virtual, superando las limitaciones de una descripción únicamente en papel. Esta aproximación resulta coherente con la necesidad de enseñar arquitectura no solo como teoría abstracta, sino como sistema dinámico de estados, transferencias y señales.

El datapath de una máquina microprogramada puede comprenderse como el conjunto de recursos sobre los cuales actúa la unidad de control. Incluye registros, buses, ALU, memoria, contadores, multiplexores y registros especiales que permiten conservar instrucciones, direcciones o datos. La microinstrucción especifica qué recursos se activan y cómo se conectan durante un ciclo o una fase de ejecución. En una organización didáctica, una microinstrucción puede indicar que el contenido de un registro se coloque en un bus, que la ALU realice una suma, que el resultado se escriba en otro registro y que el microcontador avance hacia la siguiente microinstrucción. El valor de esta representación es que permite ver el procesador como una red de transferencias controladas. Lo que en lenguaje ensamblador parece una sola operación, en microprogramación aparece como una secuencia de movimientos precisos.

La memoria de control ocupa un lugar central en este enfoque. En ella se almacenan las microinstrucciones que implementan el repertorio de instrucciones de la máquina. Cada microinstrucción contiene campos que codifican señales para el datapath y, en muchos diseños, campos para determinar la siguiente dirección microprogramada. Esta estructura permite implementar secuencias lineales, bifurcaciones internas, microbucles o decisiones basadas en banderas. De manera análoga al programa principal, el microprograma posee su propio flujo de control, pero opera en un nivel inferior. El contador de microprograma no apunta a instrucciones del usuario, sino a microinstrucciones dentro de la memoria de control. Esta distinción muestra que un procesador puede contener programas dentro del hardware, aunque estos no sean visibles para el usuario común.

La microprogramación también ayuda a explicar por qué las arquitecturas CISC encontraron en ella una herramienta históricamente conveniente. Cuando una ISA contiene instrucciones complejas, con múltiples modos de direccionamiento o efectos internos variados, implementar cada instrucción mediante control cableado puede aumentar la complejidad del diseño. Un microprograma permite describir esas instrucciones como secuencias de pasos más simples, reutilizando microoperaciones y señales existentes. Así, una instrucción compleja puede ser descompuesta en una rutina interna de control. Este enfoque favorece compatibilidad y flexibilidad, aunque puede introducir una latencia mayor que el control cableado en instrucciones simples. Por ello, la elección entre control cableado y microprogramado depende del equilibrio entre complejidad de ISA, velocidad esperada, costo de diseño, facilidad de modificación y área disponible.

En procesadores RISC, la regularidad de instrucciones redujo en muchos casos la necesidad de microprogramación extensa, porque las instrucciones podían implementarse mediante control cableado relativamente simple y pipelines uniformes. Sin embargo, esto no significa que la microprogramación haya desaparecido como principio. Incluso en procesadores modernos, ciertas operaciones complejas, excepciones, actualizaciones internas, instrucciones especiales o compatibilidad heredada pueden apoyarse en secuencias internas similares a microcódigo. Además, el concepto de microprogramación sigue siendo indispensable para comprender cómo una ISA puede traducirse en operaciones internas. La diferencia es que las microarquitecturas contemporáneas combinan control cableado rápido, decodificación avanzada, micro-operaciones, ROMs de microcódigo, predicción, renombramiento y ejecución dinámica, según la necesidad de cada instrucción.

El control microprogramado también puede analizarse desde la perspectiva de la modularidad. Al separar el datapath de la secuencia de control, se facilita modificar el comportamiento sin alterar completamente la estructura física. Jacobson y Gopalakrishnan (2001) proponen una organización de control microprogramado asincrónico, denominada microengine, orientada a implementaciones específicas de aplicación, destacando la simplicidad, modularidad, microcódigo compacto y alto rendimiento como ventajas del enfoque programable frente a alternativas rígidas de control. Este aporte permite

extender la microprogramación más allá del procesador clásico y situarla en el diseño de circuitos especializados, donde el control programable puede ofrecer flexibilidad sin renunciar necesariamente a desempeño.

El caso de los microengines asincrónicos resulta especialmente interesante porque cuestiona la idea de que la microprogramación sea siempre lenta por naturaleza. En diseños sin reloj global, la coordinación se realiza mediante protocolos de petición y reconocimiento, lo que permite encadenar acciones según la disponibilidad real de los módulos. Jacobson y Gopalakrishnan (2001) sostienen que las propiedades de los circuitos self-timed permiten organizar cadenas de computación de manera eficiente, reduciendo parte de las restricciones de planificación impuestas por diseños síncronos y acercando el rendimiento de estructuras programables al de controladores cableados especializados. Esta observación amplía el alcance del control microprogramado: su eficiencia no depende solo de estar programado o cableado, sino del modelo temporal, del datapath, del grado de especialización y del modo en que se reduce la sobrecarga de búsqueda de microinstrucciones.

El control microprogramado tiene una relación directa con el concepto de microarquitectura. Mientras la ISA define las instrucciones visibles, la microarquitectura decide cómo se implementan. La microprogramación es una de las formas más explícitas de materializar esa decisión. Permite que una misma ISA sea implementada con diferentes microprogramas o que un mismo datapath soporte variaciones de comportamiento. Esto explica por qué la arquitectura de computadores no puede limitarse a estudiar instrucciones. Es necesario comprender el mecanismo de control que convierte esas instrucciones en una secuencia de eventos internos. En términos didácticos, la microprogramación ofrece una ventana privilegiada hacia ese mecanismo, porque hace visible lo que en un procesador comercial queda oculto dentro de circuitos complejos.

En los diseños contemporáneos, la unidad de control también debe coordinarse con el pipeline. Una instrucción ya no se ejecuta necesariamente como una secuencia completa antes de iniciar la siguiente, sino que sus fases se solapan con las de otras instrucciones. Esto obliga a que las señales de control se distribuyan por etapas y se conserven en registros intermedios. En un pipeline de cinco

etapas, una instrucción puede requerir ciertas señales en ID, otras en EX, otras en MEM y otras en WB. La unidad de control debe generar y transportar esas señales de forma que cada etapa actúe correctamente en el ciclo correspondiente. Si aparece un hazard, la lógica de control debe insertar stalls, activar forwarding, cancelar escrituras o producir flush. Por tanto, el control moderno no solo decodifica instrucciones, sino que administra contingencias temporales.

La microprogramación tradicional, basada en secuencias ordenadas de microinstrucciones, puede parecer menos compatible con pipelines agresivos, superscalar y ejecución fuera de orden. Sin embargo, su principio de descomposición sigue vigente. Los procesadores modernos descomponen instrucciones en operaciones internas, aunque no siempre las ejecuten mediante una ROM microprogramada clásica. En x86, por ejemplo, ciertas instrucciones pueden traducirse en micro-operaciones que luego ingresan a un motor de ejecución dinámico. En este sentido, la microprogramación como técnica histórica se transforma en una idea más amplia: la instrucción visible puede ser una abstracción que oculta una secuencia interna de operaciones más elementales. Esta idea resulta esencial para comprender la convergencia contemporánea entre RISC, CISC y microarquitecturas híbridas.

La unidad de control también participa en la gestión de excepciones e interrupciones. Una instrucción puede producir una condición excepcional, como división por cero, fallo de página, acceso inválido o interrupción externa. El control debe preservar el estado correcto, transferir la ejecución a una rutina apropiada y garantizar que el programa pueda continuar o finalizar de manera coherente. En pipelines y ejecución fuera de orden, este problema se vuelve más delicado porque varias instrucciones pueden estar en vuelo simultáneamente. El control debe distinguir entre resultados especulativos y estado comprometido. Aunque este nivel será desarrollado con más profundidad en el subtema de ejecución fuera de orden, conviene anticipar que la unidad de control moderna no solo activa señales, sino que preserva la semántica arquitectónica frente a eventos internos y externos.

La enseñanza mediante simuladores microprogramados permite observar esta relación entre control y estado. Cuando el estudiante ejecuta microinstrucciones

paso a paso, puede ver que cada ciclo modifica registros, buses, memoria o banderas. Esta experiencia ayuda a comprender por qué una instrucción de alto nivel necesita múltiples pasos y por qué el orden de esos pasos importa. Heinikoski (2023) organiza la simulación de una máquina microprogramada alrededor de componentes como registros, buses, memoria, ALU y reloj, mostrando que la comprensión de la microprogramación requiere seguir los cambios de estado de la máquina durante la ejecución. En un libro universitario, esta perspectiva resulta valiosa porque conecta teoría, visualización y diseño.

La unidad de control microprogramada también puede vincularse con el diseño de procesadores en FPGA. En estos entornos, el control puede implementarse mediante lógica cableada, máquinas de estados o memorias que almacenan secuencias de control. La decisión depende de recursos disponibles, frecuencia deseada, facilidad de modificación y complejidad del conjunto de instrucciones. Jackson (2007), al diseñar un microprocesador serial para FPGA, discute alternativas de unidad de control, uso de recursos lógicos y variaciones de diseño, lo que evidencia que en plataformas reconfigurables el control no es una decisión secundaria, sino parte central del compromiso entre área, velocidad y flexibilidad.

En sistemas específicos de aplicación, la microprogramación puede ofrecer una vía intermedia entre procesadores generales y hardware completamente fijo. Un procesador general es flexible, pero puede desperdiciar energía y ciclos en cargas especializadas. Un circuito cableado específico puede ser muy eficiente, pero poco adaptable. Una unidad microprogramada especializada permite programabilidad limitada sobre un datapath diseñado para el dominio. Esta solución puede resultar adecuada en control embebido, procesamiento de señales, protocolos, aceleradores o dispositivos que requieren modificar comportamientos sin rediseñar todo el chip. La clave está en definir cuánta flexibilidad se necesita y cuánto costo de control puede aceptarse.

No obstante, la microprogramación también presenta limitaciones. La búsqueda de microinstrucciones consume tiempo y energía; la memoria de control ocupa área; las secuencias largas pueden aumentar latencia; y una organización demasiado flexible puede ser menos eficiente que una solución cableada

optimizada. Por ello, no debe idealizarse. Su valor depende del contexto. En una instrucción frecuente y simple, un control cableado puede ser preferible. En una operación compleja, rara o sujeta a cambios, el microprograma puede ofrecer ventajas de flexibilidad. En una arquitectura de enseñanza, su valor explicativo puede superar cualquier comparación de rendimiento. En una aplicación específica asincrónica, puede aproximarse al rendimiento de soluciones cableadas si el datapath y el control se diseñan conjuntamente.

La unidad de control, sea cableada o microprogramada, cumple finalmente una función de mediación. Media entre ISA y datapath, entre programa y señales, entre secuencia lógica y temporalidad física. En el microprocesador moderno, esta mediación se vuelve cada vez más compleja porque las instrucciones no solo se ejecutan, sino que se predicen, se transforman, se adelantan, se especulan y se comprometen en orden arquitectónico. La microprogramación permite comprender la raíz de esa mediación, incluso cuando los procesadores actuales utilizan mecanismos más avanzados. Al estudiar microprogramas, microinstrucciones y control, se aprende que el rendimiento no surge únicamente de la ALU ni de la frecuencia, sino de la capacidad de coordinar con precisión cada movimiento interno de la máquina.

La relación entre unidad de control y datapath permite observar que la microprogramación no opera en el vacío. Un microprograma solo puede activar recursos que existen físicamente en la organización del procesador. Si el datapath no posee una ruta entre dos registros, una microinstrucción no puede transferir directamente datos entre ellos; si la ALU no soporta una operación determinada, el microprograma debe construirla mediante pasos más elementales o el diseño debe incorporar una unidad adicional. Esta dependencia muestra que la flexibilidad microprogramada no sustituye al diseño físico, sino que lo presupone. El diseñador debe decidir qué recursos incluir, qué señales exponer al microcódigo, qué campos tendrá cada microinstrucción y qué grado de libertad se permitirá en la conexión entre módulos. Jacobson y Gopalakrishnan (2001) destacan que, en estructuras microprogramadas específicas de aplicación, la posibilidad de adaptar la topología del datapath y controlar el grado de programabilidad permite aproximar el rendimiento a soluciones más rígidas, sin perder completamente la capacidad de modificación.

Desde esta perspectiva, una unidad de control microprogramada puede entenderse como una arquitectura de compromiso. Si el microcódigo es muy detallado, cada microinstrucción activa pocas señales y el control resulta simple, pero el número de ciclos por instrucción puede aumentar. Si el microcódigo es muy ancho y permite activar muchas operaciones simultáneas, se reduce la cantidad de microinstrucciones necesarias, pero crecen el tamaño de la memoria de control, la complejidad del datapath y la dificultad de verificar combinaciones válidas. Esta tensión es semejante a la que existe entre procesadores escalares y VLIW: más control explícito por instrucción puede aumentar paralelismo, pero también exige mayor capacidad de planificación y una representación más amplia. En microprogramación, el diseñador decide cuánto trabajo debe realizar cada microinstrucción y cuánta complejidad puede absorber el hardware.

La anchura de la microinstrucción es una decisión técnica importante. En un microcódigo horizontal, los campos de control pueden activar directamente múltiples señales del datapath, lo que permite alto paralelismo interno y menor decodificación, aunque a costa de microinstrucciones más largas. En un microcódigo vertical, las señales están más codificadas y requieren decodificación adicional, lo que reduce el tamaño de la memoria de control, pero puede aumentar la latencia o limitar la activación simultánea de recursos. Ninguna alternativa es universalmente superior. Un diseño orientado a rendimiento puede preferir microinstrucciones más anchas para reducir pasos; uno orientado a área puede preferir codificación más compacta. La microprogramación, por tanto, reproduce dentro del control una tensión similar a la de la ISA: compacidad, velocidad, flexibilidad y costo deben equilibrarse.

El secuenciamiento de microinstrucciones también constituye una dimensión central. Una unidad microprogramada necesita determinar cuál será la siguiente microinstrucción después de la actual. En secuencias simples, basta con incrementar el microcontador; en operaciones condicionales, se requiere seleccionar entre varias direcciones según banderas, bits de estado o condiciones del datapath. Algunas arquitecturas incorporan campos de salto microprogramado, llamadas a microrutinas o tablas de despacho que permiten dirigir el flujo según el opcode de la instrucción. Este mecanismo crea un nivel de control interno análogo al programa principal, pero orientado a coordinar el

hardware. La ejecución del programa del usuario depende, en ese nivel inferior, de un programa de control que organiza la materialidad de cada instrucción.

Esta estructura resulta especialmente útil para explicar instrucciones complejas. Una instrucción de multiplicación, división, llamada a subrutina, acceso con direccionamiento indirecto o manejo de interrupción puede requerir múltiples microoperaciones coordinadas. En un control cableado, cada caso especial incrementa la complejidad de la lógica. En una unidad microprogramada, estas variantes pueden implementarse como rutas alternativas dentro del microprograma. Esta flexibilidad permite mantener la ISA visible mientras se ajusta la implementación interna. De hecho, una misma instrucción arquitectónica puede cambiar su secuencia microprogramada entre generaciones de procesadores sin afectar el código ejecutable, siempre que preserve el comportamiento observable. Esta separación entre comportamiento visible e implementación interna ha sido clave en la evolución de familias compatibles.

No obstante, la microprogramación no elimina la necesidad de optimización. Si una instrucción frecuente depende de una microrutina larga, el rendimiento global puede deteriorarse. En consecuencia, el diseñador debe identificar qué instrucciones justifican una implementación directa más rápida y cuáles pueden resolverse mediante secuencias microprogramadas más flexibles. Este criterio se relaciona con el principio de optimizar el caso común. En procesadores de propósito general, las instrucciones más frecuentes deben ejecutarse con mínima penalización; las menos frecuentes pueden tolerar mayor latencia si ello reduce complejidad de hardware. La unidad de control se convierte así en un espacio donde se distribuyen prioridades: velocidad para lo común, flexibilidad para lo complejo y corrección para todos los casos.

El microcódigo también puede servir como mecanismo de corrección o actualización. En ciertos procesadores, errores internos pueden mitigarse mediante actualizaciones de microcódigo que modifican la forma en que algunas instrucciones o condiciones especiales son gestionadas. Este hecho muestra que la microprogramación no pertenece únicamente a la historia de la arquitectura clásica, sino que conserva relevancia en sistemas contemporáneos donde la complejidad del hardware hace difícil prever todos los casos durante el diseño

inicial. La posibilidad de introducir ajustes posteriores, aunque limitada y controlada, ofrece una capa de resiliencia frente a defectos o comportamientos no previstos. En términos de ingeniería, esta capacidad tiene valor estratégico porque reduce el costo de ciertos errores tardíos.

La unidad de control microprogramada también puede interpretarse como una herramienta de compatibilidad. En arquitecturas con larga historia, mantener soporte para instrucciones heredadas puede ser costoso si todas deben implementarse con lógica dedicada. El microcódigo permite preservar comportamientos antiguos o poco frecuentes sin comprometer completamente el diseño del núcleo de ejecución más común. Esto resulta especialmente relevante en arquitecturas con fuerte legado de software, donde eliminar instrucciones podría romper compatibilidad binaria. Así, la microprogramación opera como una capa de traducción histórica: permite que nuevas microarquitecturas sostengan repertorios heredados, aunque internamente los procesen mediante secuencias más adaptadas al hardware actual.

En sistemas asincrónicos o específicos de aplicación, el control microprogramado adquiere otra dimensión. La ausencia de un reloj global permite que las operaciones avancen según la disponibilidad real de los módulos, mediante protocolos de sincronización local. Esta condición puede reducir esperas impuestas por el peor caso de un ciclo de reloj fijo, pero exige diseñar cuidadosamente la comunicación entre memoria de control, datapath y señales de reconocimiento. Jacobson y Gopalakrishnan (2001) muestran que la microprogramación asincrónica puede aprovechar encadenamientos de acciones y organización modular del datapath para reducir sobrecarga de control, lo que abre una vía intermedia entre procesadores generales y controladores completamente cableados.

La microprogramación se relaciona además con los lenguajes de descripción de hardware y con las metodologías de diseño. Cuando se diseña un procesador en VHDL, Verilog, Chisel u otros entornos, el control puede expresarse como máquinas de estados, tablas, microsecuencias o lógica combinacional. En plataformas reconfigurables, esta decisión afecta directamente el uso de LUTs, flip-flops, bloques de memoria, frecuencia máxima y facilidad de depuración.

Jackson (2007) evidencia que las decisiones sobre unidad de control, registros, ALU y rutas internas en un procesador para FPGA se reflejan en métricas concretas de uso de lógica, frecuencia y desempeño por algoritmo, lo cual confirma que el control debe evaluarse en relación con recursos físicos y carga de trabajo.

El vínculo con Chisel y las metodologías ágiles de hardware resulta pertinente para comprender la evolución actual del diseño. La construcción de procesadores abiertos y extensibles, como ciertas implementaciones RISC-V, requiere modificar decodificación, control, ALU, pipeline y toolchain de forma coordinada. Sultanpuri (2025) muestra que integrar una extensión P en un procesador RISC-V no consiste únicamente en añadir operaciones aritméticas, sino en adaptar rutas de control, etapas del pipeline, ensamblador, desensamblador y pruebas de validación para que las nuevas instrucciones sean reconocidas y ejecutadas correctamente. Este ejemplo confirma que la unidad de control sigue siendo el punto donde una extensión de ISA se vuelve realidad microarquitectónica.

En términos conceptuales, la unidad de control microprogramada permite entender que una instrucción posee dos niveles de significado. El primero es el significado arquitectónico, visible para el programador: sumar, cargar, comparar, saltar, multiplicar, convertir, transferir. El segundo es el significado organizacional, invisible para la mayoría de usuarios: activar registros, seleccionar buses, definir entrada de la ALU, habilitar escritura, evaluar banderas, actualizar direcciones internas. El microprograma traduce el primer significado en el segundo. Esta traducción muestra que la arquitectura de computadores es una disciplina de mediaciones, donde cada capa oculta y organiza la complejidad de la capa inferior.

La comparación con la segmentación instruccional también resulta enriquecedora. En un pipeline, las señales de control deben viajar junto con la instrucción a través de etapas sucesivas. En una unidad microprogramada clásica, las microinstrucciones secuencian pasos internos de una instrucción. Cuando ambos enfoques se combinan, el diseño debe decidir si ciertas instrucciones se ejecutan como flujos regulares de pipeline o si requieren entrar

en secuencias internas especiales. Las instrucciones simples pueden avanzar por la ruta común; las complejas pueden invocar microsecuencias. Esta coexistencia refleja la arquitectura híbrida de muchos procesadores modernos: una vía rápida para operaciones frecuentes y una vía controlada para casos especiales.

La ejecución fuera de orden complejiza aún más este panorama. Si las instrucciones se decodifican en micro-operaciones, estas pueden ser renombradas, reordenadas, emitidas a distintas unidades y finalmente comprometidas en orden arquitectónico. Aunque este mecanismo ya no se describe como microprogramación clásica, conserva la idea de que la instrucción visible es descompuesta en acciones internas más pequeñas. La diferencia es que, en lugar de ejecutarse como una secuencia fija, esas acciones pueden circular por una maquinaria dinámica que busca paralelismo y oculta latencias. Por ello, la microprogramación puede verse como antecedente conceptual de la descomposición interna moderna, aunque las técnicas actuales sean más agresivas y dependan de estructuras de planificación dinámica.

La unidad de control también tiene implicaciones en el consumo energético. Generar señales, acceder a memoria de control, decodificar campos, activar rutas y coordinar secuencias consume energía. Un control microprogramado flexible puede aumentar accesos internos; un control cableado complejo puede aumentar lógica activa y área. En ambos casos, el diseño debe minimizar actividad innecesaria. Las técnicas modernas de clock gating, power gating y control selectivo de unidades funcionales dependen de una unidad de control capaz de activar únicamente los recursos requeridos. Así, el control no solo organiza corrección y rendimiento, sino también eficiencia energética. Una señal mal administrada puede activar hardware innecesario y aumentar consumo sin producir trabajo útil.

Desde el punto de vista de la confiabilidad, el control es una zona crítica porque un error de secuenciamiento puede afectar múltiples instrucciones. Una ALU defectuosa puede alterar una operación específica; una señal de control mal generada puede escribir en un registro equivocado, acceder a memoria indebidamente o actualizar incorrectamente el contador de programa. Por ello, la verificación de la unidad de control es esencial. En diseños microprogramados,

se debe validar que cada microrutina produzca el comportamiento arquitectónico esperado y que las transiciones condicionales sean correctas. En diseños cableados o dinámicos, se deben verificar estados, hazards, excepciones, interrupciones y condiciones especulativas. La unidad de control es, en este sentido, el centro de la corrección operacional del procesador.

La microprogramación también mantiene valor pedagógico porque permite construir una visión gradual de la complejidad. Antes de estudiar ejecución fuera de orden, predicción avanzada o renombramiento de registros, el estudiante puede comprender cómo una instrucción se traduce en transferencias elementales. Esta base facilita luego entender por qué un procesador superscalar necesita descomponer, ordenar y preservar estado. Si no se comprende la ejecución microoperacional básica, la arquitectura moderna puede aparecer como una colección de técnicas desconectadas. En cambio, desde la microprogramación se advierte que todas esas técnicas buscan administrar el mismo problema fundamental: ejecutar instrucciones visibles mediante acciones internas correctas, rápidas y coordinadas.

En el contexto del libro, este subtema cumple una función de puente. El pipeline mostró cómo las instrucciones se solapan temporalmente; RISC y CISC mostraron cómo se define el lenguaje visible del procesador; la microprogramación muestra cómo ese lenguaje se convierte en control interno. El siguiente subtema, dedicado a la predicción de saltos y la ejecución fuera de orden, profundizará en lo que ocurre cuando el control ya no se limita a secuenciar instrucciones, sino que intenta anticipar el futuro del programa y reorganizar dinámicamente su ejecución. Desde esta progresión, la unidad de control microprogramada permite comprender que todo paralelismo avanzado descansa sobre una condición previa: la capacidad del hardware para traducir instrucciones en acciones internas verificables.

En síntesis, las unidades de control microprogramadas permiten analizar la arquitectura del microprocesador desde un nivel intermedio entre software y hardware. Su valor histórico radica en haber simplificado la implementación de instrucciones complejas; su valor técnico se encuentra en la flexibilidad, compatibilidad y modularidad; su valor educativo reside en hacer visible la

relación entre microinstrucciones, registros, buses, memoria y ALU; y su valor contemporáneo persiste en la idea de descomponer instrucciones en operaciones internas adaptadas a la microarquitectura. Aunque muchos procesadores actuales combinan control cableado, micro-operaciones, secuencias internas y planificación dinámica, la lógica microprogramada sigue siendo indispensable para comprender cómo el procesador convierte una instrucción simbólica en una ejecución física organizada.

Predicción de saltos y ejecución fuera de orden

La predicción de saltos y la ejecución fuera de orden representan dos de las estrategias más sofisticadas mediante las cuales el microprocesador moderno intenta sostener el paralelismo instruccional frente a la irregularidad real de los programas. En un pipeline ideal, las instrucciones ingresarían de forma continua, avanzarían sin interrupciones por cada etapa y producirían resultados en un flujo estable. Sin embargo, los programas contienen bifurcaciones, dependencias, accesos a memoria de latencia variable, llamadas a procedimientos, bucles y condiciones que impiden que el procesador conozca siempre con certeza cuál será la siguiente instrucción útil. La predicción de saltos surge para enfrentar la incertidumbre del control; la ejecución fuera de orden aparece para aprovechar instrucciones disponibles aunque otras anteriores estén detenidas. Ambas técnicas comparten una finalidad: impedir que las unidades funcionales permanezcan ociosas cuando existe trabajo potencialmente aprovechable.

La dependencia de control se produce cuando el flujo futuro de instrucciones depende del resultado de una rama condicional. Si el procesador espera hasta conocer el resultado del salto, el pipeline puede quedarse sin instrucciones útiles durante varios ciclos. Esta espera es especialmente costosa en procesadores superscalar, porque varias unidades de ejecución pueden quedar desalimentadas simultáneamente. La predicción de saltos intenta resolver este problema anticipando si una rama será tomada o no y cuál será la dirección probable de continuación. Pizzol, Pilla y Navaux (s. f.) explican que los procesadores superscalar aumentan el rendimiento mediante ejecución concurrente, pero dependen de alimentar el pipeline con muchas instrucciones por ciclo, por lo que

las dependencias de control reducen el tamaño efectivo de los bloques básicos y disminuyen el flujo hacia las unidades de ejecución.

La predicción puede entenderse como una forma de especulación controlada. El procesador actúa antes de tener certeza, guiándose por patrones históricos, reglas estáticas o estructuras dinámicas de hardware. Si la predicción acierta, la ejecución continúa sin penalización importante y se preserva el flujo de instrucciones. Si falla, las instrucciones ejecutadas por el camino incorrecto deben descartarse y el estado del procesador debe restaurarse o corregirse para continuar por la ruta válida. La ganancia potencial es alta porque evita ciclos vacíos; el costo también lo es porque cada mala predicción consume tiempo, energía y recursos en trabajo inútil. Por ello, la predicción de saltos no debe analizarse únicamente como una técnica de aumento de velocidad, sino como una apuesta microarquitectónica entre anticipación y recuperación.

Los esquemas de predicción más simples pueden ser estáticos. Un procesador puede asumir, por ejemplo, que los saltos no serán tomados, o que los saltos hacia atrás, frecuentes en bucles, serán tomados. Estas reglas tienen bajo costo de hardware, pero no se adaptan a patrones complejos de ejecución. La predicción dinámica, en cambio, utiliza información del comportamiento previo de las ramas. Tablas de historial, contadores saturados, buffers de destino de salto y mecanismos correlacionados permiten aprender tendencias del programa durante la ejecución. La lógica subyacente es que muchos programas exhiben regularidades: los bucles suelen repetir decisiones, las condiciones de control tienden a comportarse de manera estable durante intervalos y ciertas ramas dependen de patrones anteriores. El predictor convierte esa regularidad en una hipótesis sobre el futuro inmediato del flujo de instrucciones.

En procesadores superscalar, la predicción de saltos tiene impacto directo sobre el IPC, es decir, sobre el número de instrucciones completadas por ciclo. Un predictor ineficiente reduce el volumen de instrucciones válidas que entran al pipeline; un predictor preciso sostiene la ocupación de las unidades funcionales. Pizzol et al. (s. f.) analizan mediante simulación el efecto de diferentes niveles de precisión de predicción sobre benchmarks SPEC2000, mostrando que la precisión del predictor influye en el desempeño final de arquitecturas superscalar

y que, en ciertas condiciones, aumentar recursos de hardware puede producir resultados comparables a mejorar la precisión del predictor. Esta observación es relevante porque muestra que el rendimiento no depende de una sola técnica: puede mejorarse prediciendo mejor, ampliando recursos o equilibrando ambas estrategias según costo y beneficio.

La penalización por mala predicción aumenta conforme el pipeline se vuelve más profundo. En un pipeline corto, una rama mal predicha puede invalidar pocas instrucciones; en uno profundo, muchas etapas pueden contener trabajo de la ruta incorrecta. Además, en arquitecturas de emisión múltiple, la cantidad de instrucciones especulativas descartadas puede ser mayor. Zhou, Önder y Carr (2005) señalan que las tendencias hacia pipelines más profundos y ventanas de instrucciones más grandes incrementan la importancia de la penalización por mala predicción, debido a que el tiempo requerido para recuperar el estado correcto puede afectar significativamente el rendimiento global. De esta forma, la misma estrategia que permite aumentar frecuencia o paralelismo también intensifica el costo de equivocarse.

La recuperación ante misprediction es, por tanto, tan importante como la predicción misma. Un procesador no puede limitarse a adivinar; debe contar con mecanismos para detectar el error, descartar instrucciones incorrectas, restaurar asignaciones de registros, recuperar el contador de programa correcto y reanudar la ejecución con el menor costo posible. En procesadores sencillos, esto puede implicar vaciar el pipeline y reiniciar desde la dirección correcta. En procesadores fuera de orden, el problema es mucho más complejo porque varias instrucciones especulativas pueden haber sido renombradas, emitidas, ejecutadas o estar esperando resultados. La recuperación debe garantizar que ningún efecto de la ruta incorrecta se vuelva visible en el estado arquitectónico final.

Existen diferentes estrategias de recuperación. El checkpointing conserva copias del estado de mapeo o del estado arquitectónico en ciertos puntos, como ramas especulativas, para restaurarlas rápidamente si la predicción falla. Su ventaja es la rapidez de recuperación; su limitación está en el costo de almacenamiento y en el número de checkpoints que pueden mantenerse. Otra estrategia reconstruye el estado procesando instrucciones en orden hasta alcanzar un punto seguro, lo que

puede reducir hardware, pero aumentar el tiempo de recuperación. Zhou et al. (2005) explican que los enfoques basados en checkpointing pueden recuperar rápido, pero tienen costos de almacenamiento, mientras que los mecanismos reconstructivos escalan mejor en ciertos diseños, aunque pueden retrasar el reinicio de renombramiento y búsqueda de instrucciones correctas.

La propuesta de Eager Misprediction Recovery permite comprender cómo la arquitectura intenta ocultar parte de la penalización. En lugar de detener completamente el front-end hasta restaurar todo el estado, el procesador puede iniciar la búsqueda y renombramiento de instrucciones por la ruta correcta mientras repara valores especulativos. Las instrucciones que dependen de valores incorrectos esperan, pero aquellas que solo requieren valores correctos pueden avanzar. Zhou et al. (2005) plantean que este enfoque superpone recuperación con ejecución útil, reduciendo la penalización frente a mecanismos tradicionales de recuperación secuencial. La idea es conceptualmente poderosa: no se trata solo de recuperarse más rápido, sino de convertir una parte del tiempo de recuperación en tiempo productivo.

La ejecución fuera de orden responde a una dificultad complementaria. Aunque el programa tenga un orden lógico, no todas las instrucciones dependen estrictamente de las anteriores. Si una instrucción se detiene por un dato no disponible, como un acceso a memoria, instrucciones posteriores independientes podrían ejecutarse sin alterar el resultado final. En un procesador en orden, estas instrucciones quedarían bloqueadas; en uno fuera de orden, pueden adelantarse. Para lograrlo, el hardware debe identificar dependencias verdaderas, evitar dependencias falsas, asignar recursos dinámicamente, conservar resultados temporales y asegurar que el estado visible se actualice en el orden correcto. La ejecución fuera de orden separa el orden de emisión y ejecución interna del orden arquitectónico de finalización.

El renombramiento de registros es una técnica esencial para hacer posible esta separación. Muchas dependencias aparentes entre instrucciones no son dependencias reales de datos, sino dependencias de nombre. Por ejemplo, dos instrucciones pueden escribir en el mismo registro arquitectónico en momentos distintos sin que exista una dependencia de valor entre ellas. Si ambas usan el

mismo nombre, el procesador podría imponer restricciones innecesarias. El renombramiento asigna registros físicos distintos a ocurrencias diferentes de registros arquitectónicos, eliminando conflictos WAR y WAW, y permitiendo que instrucciones independientes avancen. Esta técnica amplía el paralelismo disponible al liberar al hardware de dependencias artificiales creadas por el número limitado de nombres visibles en la ISA.

Las estaciones de reserva, colas de emisión y ventanas de instrucciones permiten mantener instrucciones pendientes hasta que sus operandos estén disponibles y sus unidades funcionales libres. En lugar de seguir estrictamente el orden del programa, el procesador selecciona instrucciones listas para ejecutarse. Este mecanismo convierte la ejecución en un proceso de planificación dinámica. La ventana de instrucciones funciona como un espacio de oportunidad: cuanto mayor es, más posibilidades tiene el procesador de encontrar instrucciones independientes; pero también aumentan área, energía, complejidad de búsqueda, puertos y lógica de selección. Jesshope y Luo (2000) advierten que la búsqueda de mayor IPC en microprocesadores RISC mediante superpipelining, emisión múltiple, planificación dinámica, predicción y especulación incrementa considerablemente la complejidad y no siempre produce ganancias proporcionales al ancho del pipeline.

El reorder buffer o estructura equivalente cumple una función de corrección arquitectónica. Aunque las instrucciones puedan ejecutarse fuera de orden, sus resultados deben hacerse visibles en orden para preservar la semántica del programa y manejar excepciones precisas. Una instrucción ejecutada tempranamente puede producir un resultado válido, pero no debe comprometerlo definitivamente si una instrucción anterior genera una excepción o si una rama previa resulta mal predicha. El commit en orden permite que la máquina se comporte externamente como si hubiera ejecutado el programa secuencialmente, aunque internamente haya reordenado operaciones. Esta separación entre ejecución interna y estado externo es una de las claves del microprocesador moderno.

La ejecución especulativa se ubica en la intersección entre predicción de saltos y ejecución fuera de orden. El procesador no solo reordena instrucciones

independientes ya conocidas, sino que puede ejecutar instrucciones de un camino futuro aún no confirmado. Esta especulación amplía el paralelismo disponible, pero incrementa el costo de recuperación y el consumo de trabajo descartable. Si la predicción es correcta, la especulación parece haber convertido tiempo incierto en progreso real. Si es incorrecta, se debe revertir o invalidar el trabajo realizado. La microarquitectura contemporánea vive, por tanto, bajo una tensión permanente: cuanto más agresivamente especula, mayor rendimiento potencial obtiene, pero también mayor es su exposición a penalizaciones, consumo energético y complejidad de seguridad.

Las dependencias de memoria constituyen otro límite relevante para la ejecución fuera de orden. A diferencia de los registros, donde las dependencias pueden detectarse con relativa claridad mediante nombres arquitectónicos y físicos, las direcciones de memoria pueden no conocerse hasta etapas posteriores. Una carga podría depender de una tienda anterior si ambas acceden a la misma dirección. Si el procesador adelanta la carga incorrectamente, podría leer un valor obsoleto. Para sostener rendimiento, los procesadores emplean colas de carga y almacenamiento, predicción de dependencias de memoria, reenvío desde stores pendientes y mecanismos de recuperación. El problema es que la memoria introduce incertidumbre semántica adicional: no basta con saber qué instrucción viene antes, sino qué ubicación concreta de memoria será afectada.

La predicción y la ejecución fuera de orden también afectan la eficiencia energética. Cada predictor, tabla, ventana, cola, estación de reserva, puerto y registro físico consume energía. Además, la ejecución especulativa puede activar unidades funcionales para instrucciones que luego serán descartadas. La eficiencia del procesador depende de cuántas de esas operaciones producen trabajo útil. En entornos donde la energía es crítica, una microarquitectura menos agresiva puede resultar más conveniente si ofrece mejor rendimiento por watt. Esta reflexión conecta con los estudios comparativos de arquitecturas modernas, donde no basta medir tiempo de ejecución, sino energía por solución y eficiencia bajo cargas específicas. Özyılmaz (2026) subraya que los resultados entre ARM y x86-64 deben interpretarse considerando organización de núcleos, memoria, power management y características de plataforma, porque rendimiento y energía no se derivan únicamente de la ISA.

La predicción de saltos y la ejecución fuera de orden también introducen desafíos de predictibilidad temporal. En aplicaciones de propósito general, mejorar el rendimiento promedio suele justificar variabilidad. En sistemas de tiempo real, en cambio, la variabilidad puede ser problemática. Proctor y Shackleford (s. f.) indican que cachés, pipelines y ejecución especulativa pueden introducir incertidumbre temporal en microprocesadores generales usados en control, afectando el jitter y dificultando la garantía de intervalos deterministas. Esta tensión evidencia que la microarquitectura de alto rendimiento no siempre es la más apropiada para tareas donde importa más el peor caso temporal que el promedio.

La arquitectura superscalar puede entenderse como el escenario donde estas técnicas se integran. Para emitir varias instrucciones por ciclo, el procesador necesita buscar muchas instrucciones, predecir correctamente ramas, decodificar con rapidez, renombrar registros, colocar instrucciones en ventanas, seleccionar las listas, ejecutar en varias unidades funcionales y comprometer resultados en orden. Cada etapa amplía el potencial de paralelismo, pero también aumenta los puntos de fallo. Pizzol et al. (s. f.) describen el pipeline superscalar como una organización que incluye búsqueda, decodificación, despacho, emisión, ejecución y commit, junto con estructuras como colas, unidades funcionales y mecanismos de reordenamiento. Esta organización muestra que el microprocesador moderno es tanto una máquina de ejecución como una máquina de gestión de dependencias.

La recuperación ante fallos especulativos obliga a distinguir entre estado arquitectónico, estado especulativo y estado en orden. El estado arquitectónico es el que el programa debe observar; el especulativo contiene valores producidos por instrucciones aún no confirmadas; el estado en orden corresponde al punto seguro hasta el cual el programa puede considerarse comprometido. Zhou et al. (2005) analizan estas distinciones para explicar cómo los procesadores fuera de orden con especulación de control deben preservar comportamiento correcto aun cuando múltiples instrucciones se encuentren en vuelo. Esta diferenciación es fundamental porque permite que el procesador ejecute por adelantado sin permitir que los errores especulativos contaminen el resultado visible.

La complejidad de estas técnicas permite entender por qué la mejora de IPC ha encontrado límites prácticos. Ampliar el ancho de emisión, profundizar pipelines o aumentar ventanas produce beneficios decrecientes cuando las dependencias del programa, las ramas, la memoria y el consumo de control dominan. Jesshope y Luo (2000) sostienen que, aun con pipelines superscalar de cuatro u ocho vías, resulta difícil alcanzar IPC muy superiores a dos en muchos casos, debido a dependencias, penalizaciones de especulación y dificultad de mantener la tubería llena. Esta afirmación no niega la utilidad de las técnicas avanzadas, pero muestra que el paralelismo instruccional extraíble de un único hilo secuencial es limitado.

El desplazamiento hacia multithreading, multicore y manycore puede interpretarse como respuesta a ese límite. Si el paralelismo dentro de un hilo se agota o se vuelve demasiado costoso de extraer, resulta más eficiente explotar paralelismo entre hilos, procesos o datos. Sin embargo, esto no elimina la importancia de la predicción y la ejecución fuera de orden. Los núcleos de alto rendimiento todavía dependen de ellas, especialmente en cargas generales. La diferencia es que ya no se espera que un solo núcleo resuelva todo el problema del rendimiento. La arquitectura contemporánea combina núcleos eficientes, núcleos potentes, aceleradores, SIMD, GPU y paralelismo a diferentes niveles. La ejecución fuera de orden sigue siendo crucial, pero dentro de un ecosistema más amplio de paralelismo.

En síntesis, la predicción de saltos y la ejecución fuera de orden expresan el intento del hardware de anticipar y reorganizar la ejecución sin alterar el significado del programa. La predicción enfrenta la incertidumbre del flujo de control; la ejecución fuera de orden enfrenta la espera causada por dependencias y latencias; el renombramiento elimina dependencias falsas; el reorder buffer preserva el orden arquitectónico; la recuperación ante misprediction protege la corrección; y la especulación convierte probabilidad en rendimiento potencial. Estas técnicas han sido decisivas para el microprocesador moderno, pero también han incrementado complejidad, consumo, variabilidad y dificultad de verificación. Su valor debe evaluarse, por tanto, en relación con la carga de trabajo, el dominio de aplicación y el equilibrio entre rendimiento, energía y predictibilidad.

La ejecución fuera de orden también puede entenderse como una forma de tolerancia a la latencia. En un procesador estrictamente secuencial, una instrucción que espera un dato de memoria puede detener el avance de todas las instrucciones posteriores, incluso si algunas no dependen de ese dato. En un procesador fuera de orden, la ventana de instrucciones permite buscar trabajo independiente para ocupar las unidades funcionales mientras se resuelve la latencia pendiente. Esta capacidad es especialmente importante porque las diferencias de velocidad entre CPU, caché, memoria principal y almacenamiento producen esperas que no pueden eliminarse completamente. La microarquitectura intenta, entonces, ocultar la latencia mediante reorganización dinámica, del mismo modo que el pipeline intenta ocultar el tiempo total de instrucción mediante solapamiento de etapas.

Sin embargo, esta reorganización dinámica exige una infraestructura considerable. El procesador necesita mantener instrucciones en vuelo, conocer dependencias, registrar operandos disponibles, decidir qué instrucción emitir, conservar resultados temporales y garantizar que el estado arquitectónico se actualice correctamente. Cada una de estas tareas requiere estructuras de hardware: tablas de renombramiento, registros físicos adicionales, colas de instrucciones, estaciones de reserva, unidades de planificación, colas de carga y almacenamiento, y buffers de retiro. En conjunto, estas estructuras convierten al núcleo moderno en un sistema de gestión de flujo altamente complejo. La ejecución deja de ser una simple secuencia de operaciones y se convierte en una administración continua de posibilidades.

El renombramiento de registros resulta decisivo porque separa los registros arquitectónicos visibles para el software de los registros físicos utilizados internamente. Esta separación permite que varias instrucciones que usan el mismo nombre arquitectónico no bloqueen innecesariamente la ejecución si no existe una dependencia real de valor. En términos conceptuales, el renombramiento amplía el espacio interno de representación del procesador. El programa parece operar con un conjunto limitado de registros, pero la microarquitectura dispone de un conjunto más amplio para organizar versiones temporales, eliminar conflictos y aumentar paralelismo. Esta técnica evidencia

que la ISA no revela toda la capacidad interna de la máquina; el procesador moderno crea recursos invisibles para sostener una ejecución más flexible.

El reorder buffer, o una estructura equivalente, permite resolver la tensión entre ejecución interna desordenada y comportamiento externo ordenado. Aunque las instrucciones puedan ejecutarse en un orden distinto al escrito, el programa debe observar resultados como si se hubieran comprometido respetando la secuencia arquitectónica. Esta condición resulta imprescindible para manejar excepciones precisas, interrupciones, fallos de memoria y predicciones incorrectas. Si una instrucción posterior modifica el estado antes de que una anterior sea validada, el procesador podría quedar en una condición incoherente ante una excepción. Por ello, el commit ordenado funciona como una frontera de seguridad: internamente se explora paralelismo; externamente se conserva la semántica secuencial.

La predicción de saltos y la ejecución especulativa muestran, además, que el microprocesador moderno trabaja con futuros posibles. Cuando una rama no ha sido resuelta, el predictor elige una dirección probable y el front-end comienza a traer instrucciones de ese camino. Si la ruta es correcta, se ha ganado tiempo; si no lo es, la máquina debe corregir el error. Esta forma de anticipación es una de las razones por las cuales los procesadores modernos logran alto rendimiento en código secuencial. Pero también es una fuente de consumo energético y de complejidad de seguridad, porque el hardware puede ejecutar transitoriamente instrucciones que nunca debieron formar parte del flujo arquitectónico válido. Aunque este capítulo no se centra en seguridad microarquitectónica, es importante reconocer que la especulación transforma la ejecución en un proceso probabilístico administrado por hardware.

En procesadores superscalar, la predicción de saltos no solo decide una dirección; también determina si el front-end podrá mantener suficiente ancho de suministro de instrucciones. Un núcleo capaz de emitir varias instrucciones por ciclo necesita un flujo continuo de instrucciones decodificables. Una sola rama mal predicha puede cortar ese flujo y reducir la ocupación de múltiples unidades funcionales. Por eso, la predicción se vincula con el ancho del procesador. Cuanto más amplio es el núcleo, mayor es el costo de no alimentarlo. La arquitectura

superscalar, en consecuencia, depende de la precisión del predictor, de la capacidad de búsqueda, de la caché de instrucciones y de la velocidad de recuperación para convertir su ancho teórico en rendimiento efectivo.

La ejecución fuera de orden también depende de la disponibilidad de paralelismo en el propio programa. Si una secuencia posee dependencias verdaderas encadenadas, el procesador no puede inventar independencia. Puede eliminar dependencias falsas, ocultar latencias, adelantar instrucciones independientes y especular sobre ramas, pero no puede violar una dependencia real sin alterar el resultado. Esta limitación explica por qué ampliar indefinidamente ventanas de instrucciones o unidades funcionales no produce mejoras proporcionales. El paralelismo a nivel de instrucción existe, pero es finito y depende del código. La microarquitectura puede descubrirlo y explotarlo mejor o peor, pero no puede crear paralelismo donde el algoritmo impone secuencialidad estricta.

La siguiente tabla sintetiza las principales técnicas microarquitectónicas que permiten sostener el paralelismo instruccional en procesadores modernos. Su propósito es organizar de forma comparativa el problema que cada técnica atiende, la ventaja que ofrece y el costo que introduce, evitando una lectura aislada de cada mecanismo.

Tabla 7

Técnicas microarquitectónicas para sostener el paralelismo instruccional

Técnica microarquitectónica	Problema que atiende	Ventaja principal	Costo o limitación	Relación con el microprocesador moderno
Segmentación instruccional	Baja utilización de recursos cuando las fases de instrucción se ejecutan secuencialmente	Aumenta throughput al solapar IF, ID, EX, MEM y WB	Introduce hazards, registros interetapa y complejidad de control	Base del paralelismo interno en procesadores RISC y CISC modernos
Predicción de saltos	Incertidumbre sobre la próxima instrucción ante ramas condicionales	Mantiene alimentado el pipeline y reduce esperas de control	Las malas predicciones generan flush, recuperación y trabajo desperdiciado	Esencial en pipelines profundos y superscalar
Ejecución especulativa	Necesidad de avanzar antes de confirmar completamente	Convierte predicciones acertadas en progreso útil	Consume energía en instrucciones que pueden descartarse	Permite explotar rutas probables y sostener alto IPC

	el flujo de control			
Renombramiento de registros	Dependencias falsas por reutilización de nombres arquitectónicos	Elimina conflictos WAR y WAW, aumentando paralelismo disponible	Requiere registros físicos, tablas de mapeo y lógica adicional	Separa registros visibles de recursos internos de ejecución
Ejecución fuera de orden	Bloqueo de instrucciones independientes por latencias o dependencias anteriores	Permite ejecutar instrucciones listas aunque otras previas esperen	Aumenta complejidad de planificación, verificación y consumo	Núcleo de microarquitecturas de alto rendimiento
Reorder buffer y commit ordenado	Necesidad de preservar semántica secuencial y excepciones precisas	Permite ejecución interna flexible con estado externo correcto	Ocupa área, energía y puede limitar el número de instrucciones en vuelo	Garantiza corrección arquitectónica en procesadores especulativos
Checkpointing	Recuperación rápida ante saltos mal predichos	Restaura estado con baja latencia	Requiere almacenamiento proporcional a ramas o estados especulativos	Usado para reducir penalización por misprediction
Eager Misprediction Recovery	Pérdida de ciclos durante recuperación tradicional	Superpone recuperación con ejecución útil de instrucciones correctas	Requiere identificar valores especulativos y coordinar restauración fina	Reduce penalización en procesadores fuera de orden
SMT o multithreading simultáneo	Burbujas del pipeline y subutilización de unidades funcionales	Usa instrucciones de otros hilos para mejorar ocupación	Puede generar contención por caché, registros y unidades funcionales	Complementa ILP con paralelismo entre hilos
SIMD y extensiones vectoriales	Necesidad de aplicar la misma operación a múltiples datos	Incrementa throughput en datos paralelizables	Requiere datos empaquetables, alineación y soporte de compilador	Importante en DSP, multimedia, IA y cómputo científico

Nota. *Elaboración propia a partir de Pizzol et al. (s. f.), Zhou et al. (2005), Jesshope y Luo (2000), Nikolić et al. (2022), Sultanpuri (2025) y Özyılmaz (2026).*

La tabla muestra que el paralelismo instruccional no depende de una única técnica, sino de una arquitectura de mecanismos complementarios. La segmentación crea la base temporal del flujo; la predicción de saltos intenta

mantenerlo; la especulación lo extiende hacia rutas probables; el renombramiento elimina restricciones artificiales; la ejecución fuera de orden aprovecha instrucciones listas; y el commit ordenado preserva la ilusión secuencial. A su vez, técnicas como checkpointing o recuperación ansiosa buscan reducir el costo de equivocarse, mientras que SMT y SIMD amplían la explotación de paralelismo hacia hilos y datos. La microarquitectura moderna puede entenderse como la coordinación de todas estas estrategias para convertir código secuencial o parcialmente paralelo en ejecución eficiente.

La inclusión de SIMD dentro de esta discusión es relevante porque el microprocesador moderno ya no se limita al paralelismo a nivel de instrucción. Las extensiones vectoriales y packed-SIMD permiten que una sola instrucción opere sobre varios datos, ampliando el rendimiento en dominios como procesamiento digital de señales, imágenes, audio, visión computacional e inteligencia artificial. Sultanpuri (2025) evidencia este punto al implementar una ALU para la extensión P de RISC-V orientada a DSP, donde kernels como producto punto, FIR, multiplicación matricial y Sobel se benefician de reducir instrucciones retiradas y ciclos de CPU mediante datos empaquetados. Esta forma de paralelismo no reemplaza la predicción ni la ejecución fuera de orden, pero las complementa al ofrecer una vía explícita de paralelización de datos.

No obstante, SIMD también introduce condiciones. Para aprovecharlo, los datos deben organizarse de manera adecuada, los bucles deben vectorizarse, las dependencias deben permitir operaciones simultáneas y el costo de empaquetamiento no debe superar la ganancia. En cargas pequeñas, el overhead de preparar datos puede reducir el beneficio; en cargas grandes y regulares, el efecto puede ser sustancial. Esta observación conecta con una idea general del capítulo: ninguna técnica microarquitectónica produce ventajas absolutas. Todas dependen de la forma concreta del programa, del tamaño de datos, de la memoria, del compilador y de los límites físicos del sistema.

La ejecución fuera de orden también debe evaluarse frente a arquitecturas alternativas. Algunos sistemas embebidos, microcontroladores o procesadores de tiempo real prefieren ejecución en orden para reducir área, consumo y

variabilidad temporal. Aunque pierdan rendimiento promedio frente a núcleos fuera de orden, pueden ofrecer mayor predictibilidad, menor consumo y verificación más sencilla. En control industrial, automoción, sistemas médicos o aplicaciones críticas, la capacidad de estimar el peor caso temporal puede ser más importante que maximizar IPC promedio. Esta diferencia refuerza la tesis de que la mejor microarquitectura no es la más agresiva, sino la más adecuada para la finalidad del sistema.

El costo energético de estas técnicas también es considerable. Un predictor avanzado consume energía en cada acceso; una ventana grande requiere comparaciones múltiples; el renombramiento necesita tablas activas; las colas de emisión realizan búsquedas asociativas; y la especulación puede activar unidades para trabajo que luego se descarta. En plataformas móviles o portátiles, esta actividad debe controlarse mediante políticas de energía, apagado selectivo, escalado dinámico de frecuencia y voltaje, y asignación heterogénea de tareas. El rendimiento por watt se convierte entonces en un criterio central para decidir cuán agresiva debe ser la microarquitectura. No toda carga justifica activar todo el potencial del núcleo.

La relación con los compiladores también es decisiva. Aunque la predicción y la ejecución fuera de orden son técnicas de hardware, el compilador puede facilitar o dificultar su trabajo mediante organización de instrucciones, eliminación de ramas innecesarias, desenrollado de bucles, vectorización, alineación de datos y selección de instrucciones. Un código con buena localidad, ramas predecibles y dependencias reducidas permite que el hardware extraiga más paralelismo. Un código irregular, con saltos difíciles, estructuras dispersas y dependencias estrechas, limita el beneficio de la microarquitectura avanzada. En consecuencia, el rendimiento moderno surge de una cooperación entre compilador y hardware, no de una sola capa.

Las arquitecturas multihilo intentan abordar otra limitación: cuando un hilo no ofrece suficiente paralelismo o se detiene por latencia, otro hilo puede ocupar recursos. El SMT comparte unidades funcionales entre varios hilos para mejorar utilización. Sin embargo, esta estrategia también introduce competencia por caché, predictor, registros, ancho de banda y unidades de ejecución. Jesshope y

Luo (2000) analizan el SMT como una alternativa para ampliar la ventana efectiva de instrucciones mediante múltiples contextos, aunque advierten que el tamaño de archivos de registros, el acceso a recursos compartidos y la complejidad del front-end pueden convertirse en problemas de diseño. Así, incluso cuando el paralelismo se busca entre hilos, el hardware enfrenta nuevos costos de coordinación.

La ejecución fuera de orden puede verse como una forma de inteligencia local del hardware. El procesador observa una ventana de instrucciones, identifica cuáles están listas, decide cuáles emitir y conserva la apariencia secuencial. No comprende el programa en sentido semántico amplio, pero reconoce dependencias, disponibilidad de operandos y recursos. Esta inteligencia microarquitectónica permite ejecutar código legado o general sin exigir que el programador exponga todo el paralelismo. Sin embargo, su costo es elevado. En dominios donde el paralelismo es explícito y regular, puede ser más eficiente trasladar parte de esa responsabilidad al compilador, a SIMD, a GPU, a VLIW o a aceleradores especializados.

La predicción de saltos también posee un componente estadístico. No garantiza certeza, sino probabilidad. Esto la diferencia de mecanismos estrictamente deterministas de la arquitectura. Un buen predictor aprende patrones y reduce errores, pero siempre puede fallar ante comportamientos irregulares, entradas adversas o cambios de fase del programa. Por ello, el diseño del predictor implica elegir entre tamaño de tablas, complejidad de correlación, consumo y precisión esperada. Un predictor muy simple puede fallar demasiado; uno muy complejo puede consumir energía y área excesivas. La arquitectura moderna debe decidir cuánto hardware destinar a anticipar el flujo de control y cuánto a recuperarse cuando la anticipación falla.

La recuperación ante misprediction muestra una lección general de la microarquitectura: el rendimiento no depende solo de evitar errores, sino de limitar su costo. Incluso un predictor excelente fallará algunas veces. Si la recuperación es lenta, esos fallos dominan el rendimiento; si es rápida, el sistema puede tolerar cierta tasa de error. Este razonamiento se aplica también a caché, memoria, predicción de dependencias y especulación. Una arquitectura

eficiente no es aquella que elimina toda incertidumbre, sino aquella que administra sus consecuencias de manera proporcionada.

En los procesadores contemporáneos orientados a inteligencia artificial y computación científica, estas técnicas coexisten con aceleradores especializados. La CPU puede encargarse del control general, la preparación de datos y partes secuenciales, mientras GPU, NPU, DSP o unidades vectoriales ejecutan operaciones masivamente paralelas. En este ecosistema, la predicción de saltos y la ejecución fuera de orden siguen siendo relevantes para la CPU, pero ya no constituyen la única vía de rendimiento. La arquitectura moderna distribuye el trabajo según su naturaleza: control irregular en CPU, datos paralelos en SIMD o GPU, operaciones especializadas en aceleradores y tareas de baja potencia en núcleos eficientes. El microprocesador se convierte así en un coordinador dentro de un sistema heterogéneo.

Esta heterogeneidad tiene implicaciones para el diseño de software. El programador y el compilador deben decidir qué partes del algoritmo se benefician de ejecución general, cuáles de vectorización, cuáles de aceleración y cuáles de paralelismo entre hilos. La CPU fuera de orden puede ocultar latencias y explotar paralelismo fino, pero no sustituye la necesidad de rediseñar algoritmos para datos masivos. En inteligencia artificial, procesamiento de señales o simulación científica, la mayor ganancia suele provenir de reorganizar el cálculo para explotar paralelismo de datos y reducir movimiento de memoria. La microarquitectura avanzada ayuda, pero no reemplaza el diseño algorítmico consciente del hardware.

La complejidad de predicción y ejecución fuera de orden también plantea un desafío de verificabilidad. El número de estados posibles crece con instrucciones en vuelo, especulación, excepciones, dependencias de memoria, interrupciones y recuperación. Cada combinación puede producir casos límite. La verificación debe asegurar que, pese a la ejecución interna desordenada, el estado final sea correcto. En procesadores de alto rendimiento, esta tarea es una de las más costosas del diseño. En diseños académicos o abiertos, la complejidad puede limitarse para facilitar implementación y validación. Esta es otra razón por la cual

muchas arquitecturas embebidas o RISC-V experimentales optan inicialmente por pipelines más simples antes de incorporar técnicas agresivas.

Desde un punto de vista crítico, la predicción de saltos y la ejecución fuera de orden representan una etapa de madurez y, al mismo tiempo, de tensión en la evolución del microprocesador. Permitieron sostener rendimiento en código secuencial durante años, pero aumentaron complejidad, energía y dificultad de garantizar tiempos. La transición hacia multicore, manycore y aceleradores no elimina estas técnicas, pero reduce su centralidad exclusiva. El futuro del rendimiento probablemente no dependerá solo de hacer núcleos cada vez más especulativos y complejos, sino de combinarlos con unidades más simples, paralelismo explícito, especialización y software capaz de distribuir el trabajo de manera inteligente.

En consecuencia, la predicción de saltos y la ejecución fuera de orden deben entenderse como mecanismos de optimización bajo incertidumbre. La predicción enfrenta incertidumbre sobre el flujo; la ejecución fuera de orden enfrenta incertidumbre sobre disponibilidad de datos y latencias; la recuperación enfrenta incertidumbre sobre errores especulativos; y el commit ordenado preserva certeza arquitectónica. Esta combinación define una de las paradojas del microprocesador moderno: para comportarse externamente como una máquina secuencial precisa, internamente debe operar como una máquina especulativa, dinámica y probabilística. Esa paradoja será el punto de partida de la discusión crítica, centrada en la tensión entre eficiencia energética y rendimiento computacional en las microarquitecturas de última generación.

Discusión crítica

La eficiencia energética frente al rendimiento computacional constituye una de las tensiones más complejas de las microarquitecturas contemporáneas, porque el microprocesador moderno ya no puede mejorar indefinidamente su desempeño mediante el aumento de frecuencia, la profundización del pipeline o la ampliación de unidades funcionales. Durante décadas, el diseño de procesadores se orientó hacia la extracción agresiva de paralelismo a nivel de instrucción mediante segmentación, emisión múltiple, predicción de saltos,

ejecución especulativa, renombramiento de registros y ejecución fuera de orden. Estas técnicas permitieron sostener incrementos notables de rendimiento en programas secuenciales, pero también trasladaron el problema hacia una zona de creciente complejidad energética. Cada predictor, tabla de renombramiento, buffer de reordenamiento, cola de emisión, estación de reserva, unidad funcional adicional y mecanismo de recuperación ante fallos especulativos consume área, potencia dinámica y energía de control. Por ello, el rendimiento del microprocesador moderno ya no puede evaluarse solo por instrucciones por ciclo o frecuencia de reloj, sino por la cantidad de trabajo útil que consigue producir por unidad de energía consumida.

La segmentación instruccional expresa con claridad esta ambivalencia. Por un lado, el pipeline permite aumentar el throughput al solapar fases de ejecución y mantener varias instrucciones activas al mismo tiempo. Por otro, introduce registros intermedios, lógica de control, mecanismos de detección de hazards, rutas de forwarding, interlocks y circuitos de flush que incrementan la complejidad del diseño. En pipelines más profundos, el procesador puede alcanzar frecuencias más elevadas, pero también crece la penalización por saltos mal predichos y la energía desperdiciada en instrucciones que deben descartarse. La paradoja es evidente: la técnica que mejora el flujo de instrucciones también aumenta la sensibilidad del sistema a interrupciones de ese flujo. Ponrani, Thanishtha y Swasthika (2026) muestran que la incorporación de forwarding e interlocks mejora la resolución de hazards en un pipeline RISC de cinco etapas, pero esa mejora exige lógica adicional y un diseño más complejo, lo cual confirma que el rendimiento del pipeline depende de una inversión microarquitectónica que no es energéticamente neutra.

La predicción de saltos intensifica esta tensión porque convierte la ejecución en una actividad anticipatoria. Mientras más profundo y ancho sea el pipeline, más costoso resulta detenerse ante una rama condicional; por ello, el procesador predice. Pero predecir implica mantener estructuras de historial, tablas, buffers de destino, mecanismos de actualización y rutas de recuperación. Cuando la predicción acierta, la energía invertida se transforma en progreso útil; cuando falla, se convierte en actividad especulativa descartada. Pizzol, Pilla y Navaux (s. f.) evidencian que la precisión de la predicción afecta el rendimiento de

procesadores superscalar, aunque también señalan que en ciertos casos ampliar hardware puede producir efectos comparables a mejorar la precisión del predictor. Esta relación plantea una pregunta crítica: hasta qué punto conviene aumentar estructuras de predicción y recursos de ejecución si los beneficios dependen de patrones de carga variables y pueden degradarse en programas con ramas irregulares.

La recuperación ante mala predicción revela otra dimensión del problema. Los mecanismos de checkpointing permiten restaurar rápidamente el estado, pero requieren almacenamiento adicional. Los enfoques reconstructivos pueden reducir ciertos costos de hardware, pero aumentan la latencia de recuperación. Las propuestas como Eager Misprediction Recovery buscan superponer recuperación con ejecución útil, pero para hacerlo necesitan identificar valores especulativos con mayor granularidad y administrar la restauración del estado con precisión. Zhou, Önder y Carr (2005) muestran que la penalización por misprediction se vuelve crítica en procesadores fuera de orden con pipelines profundos, y que reducirla exige mecanismos cada vez más refinados. El punto crítico es que el procesador moderno dedica una parte considerable de su estructura no a ejecutar instrucciones correctas, sino a sostener la posibilidad de equivocarse menos o recuperarse mejor cuando se equivoca. Esa infraestructura mejora rendimiento promedio, pero incrementa energía, área y complejidad de verificación.

La ejecución fuera de orden profundiza aún más el dilema. Su valor radica en aprovechar instrucciones independientes cuando otras se detienen por dependencias o memoria. Sin embargo, para lograrlo necesita ventanas de instrucciones, renombramiento, registros físicos adicionales, colas, estaciones de reserva, lógica de selección, buffers de reordenamiento y mecanismos de commit ordenado. Estas estructuras operan permanentemente para descubrir paralelismo que no está explícito en el código. Jesshope y Luo (2000) advierten que las técnicas orientadas a elevar el IPC, como superpipelining, emisión múltiple, predicción dinámica y especulación, no siempre producen incrementos proporcionales al ancho de la arquitectura y pueden generar penalizaciones importantes en área, tiempo y dependencia de la aplicación. Esta advertencia conserva vigencia porque muestra que el paralelismo instruccional extraído

dinámicamente tiene rendimientos decrecientes. Mientras más agresivo es el núcleo, más energía consume para encontrar oportunidades adicionales que pueden ser escasas o dependientes del comportamiento del programa.

La comparación entre RISC, CISC y arquitecturas híbridas permite problematizar la idea de que la eficiencia energética pueda derivarse únicamente de la ISA. Una ISA regular puede facilitar decodificación y pipeline, pero una implementación concreta puede ser compleja, especulativa y energéticamente costosa. Una ISA CISC puede presentar mayor complejidad visible, pero traducir instrucciones en micro-operaciones internas optimizadas y beneficiarse de décadas de refinamiento microarquitectónico. Özyılmaz (2026) advierte que las diferencias entre ARM y x86-64 deben interpretarse como resultados de plataforma, no como efectos puros de la ISA, porque intervienen memoria, organización de núcleos, integración del sistema, políticas de energía y metodología de medición. Esta precisión es fundamental para evitar conclusiones reduccionistas. La eficiencia energética no pertenece automáticamente a una familia arquitectónica; emerge de la relación entre ISA, microarquitectura, tecnología de fabricación, carga de trabajo, sistema operativo y diseño de plataforma.

En este sentido, el debate RISC/CISC se desplaza hacia una pregunta más amplia: cómo distribuir la complejidad para que produzca rendimiento útil sin exceder los límites energéticos. RISC simplifica el contrato visible de instrucción, pero puede requerir más instrucciones o extensiones especializadas para ciertas cargas. CISC conserva densidad y compatibilidad, pero requiere decodificación compleja y mecanismos internos de traducción. RISC-V propone modularidad y apertura, pero su eficiencia depende de implementaciones concretas, toolchains maduros y extensiones adecuadas. Rosdiyanto, Zuhro y Nisfuwadi (2025) sostienen que la selección entre RISC y CISC en sistemas embebidos debe considerar energía, complejidad, respuesta a interrupciones, memoria y soporte de herramientas. Este criterio resulta pertinente porque la microarquitectura actual no se elige por superioridad abstracta, sino por adecuación a restricciones específicas.

Las extensiones especializadas, como SIMD o packed-SIMD, ofrecen una respuesta parcial al costo creciente de la ejecución general. En lugar de gastar

energía descubriendo paralelismo implícito en una secuencia escalar, permiten expresar paralelismo de datos de forma más directa. Sultanpuri (2025) muestra que la implementación de la extensión P de RISC-V puede reducir ciclos e instrucciones retiradas en kernels DSP como producto punto, filtros FIR, multiplicación matricial y Sobel. Sin embargo, incluso estas extensiones presentan límites: requieren datos empaquetables, alineación, soporte de compilador y cargas suficientemente grandes para compensar el costo de preparación. La especialización mejora rendimiento por watt cuando existe correspondencia entre instrucción y patrón de datos, pero puede aportar poco si la carga es irregular, pequeña o dominada por control.

El diseño de unidades funcionales especializadas, como FPU, DSP o aceleradores en FPGA, revela otra tensión entre rendimiento, área y energía. Liang, Tessier y Mencer (2003) muestran que las unidades de punto flotante para FPGA pueden generarse con múltiples combinaciones de área, throughput y latencia, lo que evidencia que no existe una FPU óptima en abstracto, sino configuraciones adaptadas a restricciones concretas. Kwon, Sondeen y Draper (2005) también plantean que las arquitecturas de FPU deben ajustarse a metas divergentes, como minimizar área en procesamiento en memoria o maximizar throughput en aplicaciones embebidas de flujo. Estas investigaciones permiten comprender que la eficiencia energética suele requerir especialización, pero la especialización introduce pérdida de generalidad y exige decisiones cuidadosas sobre qué operaciones merecen soporte dedicado.

La computación reconfigurable refuerza esta lectura crítica. Los FPGA permiten adaptar hardware a cargas concretas, pero su eficiencia depende de cómo se utilicen recursos como LUTs, flip-flops, bloques de memoria e interconexiones. Jackson (2007) propone una arquitectura bit-serial para FPGA con el objetivo de reducir uso de lógica, aunque reconoce que ello aumenta el número de ciclos requeridos. Esta propuesta cuestiona la equivalencia entre mayor paralelismo y mejor diseño. En ciertos escenarios, una arquitectura menos ancha pero mucho más pequeña puede ofrecer mejor rendimiento por unidad de lógica o menor consumo. La eficiencia, entonces, no siempre se encuentra en maximizar paralelismo inmediato, sino en equilibrar área, frecuencia, ciclos, energía y escala de despliegue.

La microprogramación también debe leerse desde esta tensión. Una unidad de control microprogramada puede aportar flexibilidad, compatibilidad y capacidad de modificación, pero implica memoria de control, acceso a microinstrucciones y posible aumento de ciclos. En aplicaciones específicas, sin embargo, el microcódigo compacto y un datapath adaptado pueden aproximarse al rendimiento de controladores cableados. Jacobson y Gopalakrishnan (2001) plantean que los microengines asincrónicos pueden aprovechar la modularidad, el encadenamiento y la ausencia de un reloj global para obtener alto rendimiento con control programable. La lección crítica es que la programabilidad no es necesariamente sinónimo de ineficiencia, siempre que el nivel de flexibilidad se ajuste al dominio de aplicación. El exceso de generalidad desperdicia recursos; la rigidez absoluta dificulta evolución y corrección.

En sistemas de tiempo real, el rendimiento promedio puede ser menos importante que la predictibilidad. Las técnicas modernas como cachés, pipelines profundos, predicción y especulación incrementan la velocidad media, pero también introducen variabilidad temporal. Proctor y Shackleford (s. f.) analizan cómo los microprocesadores generales que ejecutan sistemas operativos en tiempo real pueden sufrir jitter por efectos de caché, pipeline, bus locking, secciones críticas e interrupciones, afectando especialmente tareas de control con períodos reducidos. Esta observación problematiza el ideal de microarquitectura agresiva. Un núcleo fuera de orden puede ser excelente para cargas generales, pero menos adecuado para control duro si dificulta estimar el peor caso. La eficiencia educativa y técnica exige distinguir entre velocidad promedio, latencia máxima, jitter y determinismo.

La heterogeneidad aparece como respuesta contemporánea a muchas de estas tensiones. En lugar de diseñar un único núcleo cada vez más complejo, las plataformas actuales combinan núcleos de alto rendimiento, núcleos eficientes, GPU, DSP, NPU, aceleradores criptográficos, unidades vectoriales y controladores especializados. Esta organización permite asignar cada tarea al recurso más adecuado. Una carga interactiva puede usar núcleos potentes; tareas de fondo pueden ejecutarse en núcleos eficientes; operaciones matriciales pueden desplazarse a GPU o SIMD; inferencia de IA puede utilizar aceleradores. No obstante, la heterogeneidad no elimina la complejidad; la desplaza hacia la

planificación, la coherencia de memoria, el movimiento de datos, el soporte de software y la programación. El sistema puede ser energéticamente superior solo si evita que el costo de coordinación supere la ganancia de especialización.

La inteligencia artificial y la computación científica intensifican esta discusión. Sus cargas suelen requerir operaciones vectoriales, matriciales, paralelismo de datos y alto ancho de banda de memoria. Una CPU general, incluso con ejecución fuera de orden, puede ser insuficiente o energéticamente ineficiente para estos patrones. Por ello, las microarquitecturas de última generación tienden a integrar extensiones vectoriales, aceleradores y memoria más cercana al cómputo. Sin embargo, una parte del trabajo sigue dependiendo de la CPU: control, orquestación, partes secuenciales, preparación de datos, gestión de memoria y coordinación del sistema. El futuro no parece orientarse a reemplazar la CPU, sino a redefinirla como coordinadora dentro de una arquitectura heterogénea donde el rendimiento surge de distribuir correctamente el trabajo.

El movimiento de datos es una limitación transversal que afecta tanto rendimiento como energía. Una unidad funcional puede ser extremadamente rápida, pero si los datos no llegan con suficiente ancho de banda o deben moverse repetidamente entre CPU, memoria, GPU o aceleradores, la ganancia se reduce. Este problema conecta el capítulo actual con el anterior: el microprocesador moderno no puede aislarse de la jerarquía de memoria ni de la interconexión. Muchas veces, la energía de mover datos supera la energía de operar sobre ellos. Por tanto, la eficiencia microarquitectónica exige diseñar instrucciones, caches, registros, buses y aceleradores de manera coherente con la localidad. La especialización computacional sin especialización del movimiento de datos produce núcleos rápidos pero subutilizados.

Desde una perspectiva epistemológica, la microarquitectura contemporánea revela una paradoja: para mantener la ilusión de una máquina secuencial precisa, el procesador ejecuta internamente de manera especulativa, paralela, desordenada y probabilística. El programa observa un orden lógico; el hardware explora múltiples posibilidades, adelanta instrucciones, descarta caminos, renombra registros y retrasa el compromiso del estado. Esta distancia entre apariencia arquitectónica y comportamiento físico ha sido productiva para el

rendimiento, pero también crea opacidad. Comprender un procesador moderno exige aceptar que la ISA es solo la superficie visible de una maquinaria interna mucho más dinámica. Esta opacidad plantea desafíos educativos, de verificación, de seguridad y de optimización.

Los límites de la especulación también deben analizarse críticamente. La especulación mejora rendimiento cuando las predicciones son acertadas y la recuperación es eficiente; pero si las cargas presentan ramas difíciles, datos irregulares o baja localidad, el hardware especulativo puede consumir energía sin obtener mejoras proporcionales. Además, la complejidad especulativa ha mostrado implicaciones de seguridad en la historia reciente de la computación, aunque este capítulo no se concentre en esa dimensión. Lo relevante para el argumento arquitectónico es que la especulación no es un bien absoluto: es una estrategia de riesgo controlado. Su conveniencia depende del equilibrio entre probabilidad de acierto, costo de error, consumo energético y criticidad del dominio.

La comparación entre microarquitecturas de alto rendimiento y diseños embebidos muestra que la eficiencia debe definirse contextual y no universalmente. Un procesador fuera de orden con predictor avanzado puede ser eficiente para escritorio, servidores o portátiles de alto desempeño. Un núcleo en orden, simple y predecible puede ser más eficiente en sensores, controladores o sistemas alimentados por batería. Una FPGA puede ser adecuada para prototipado o aceleración específica. Una extensión SIMD puede ser excelente para DSP, pero poco útil en control irregular. Un microengine programable puede ser valioso cuando se requiere flexibilidad limitada. La arquitectura eficiente es aquella que ajusta su complejidad al problema.

En consecuencia, la búsqueda de rendimiento ya no puede basarse en la acumulación indiscriminada de técnicas. Profundizar pipelines, ampliar ventanas, agregar predictores, aumentar unidades, incorporar extensiones y añadir aceleradores solo tiene sentido si la carga puede aprovecharlos y si el costo energético se mantiene dentro de límites razonables. La arquitectura contemporánea requiere una racionalidad selectiva: activar complejidad donde produce valor y evitarla donde solo añade consumo. Esta racionalidad se expresa

en núcleos heterogéneos, apagado selectivo de unidades, DVFS, instrucciones especializadas, diseño modular y aceleración específica de dominio.

La formación universitaria en arquitectura de computadores debe reflejar esta complejidad. Ya no basta enseñar la CPU como una secuencia de fetch, decode y execute; tampoco basta presentar RISC y CISC como categorías opuestas. Es necesario mostrar cómo el pipeline introduce hazards, cómo la predicción administra incertidumbre, cómo la ejecución fuera de orden preserva una apariencia secuencial, cómo la microprogramación traduce instrucciones en control y cómo la energía condiciona todas estas decisiones. El estudiante debe comprender que el microprocesador moderno es una máquina de compromisos, no una suma de mejoras lineales. Cada avance técnico abre posibilidades y produce nuevos límites.

La discusión crítica permite sostener que el problema central de las microarquitecturas de última generación no es elegir entre rendimiento y energía como si fueran polos aislados, sino diseñar una relación equilibrada entre trabajo útil, complejidad necesaria y consumo aceptable. La eficiencia energética no significa renunciar al rendimiento; significa obtener rendimiento con la menor actividad improductiva posible. Esto implica reducir especulación fallida, evitar movimiento innecesario de datos, utilizar unidades especializadas cuando corresponda, mantener predictibilidad donde sea crítica y coordinar hardware y software para que la arquitectura trabaje sobre patrones que realmente puede explotar.

En síntesis, el microprocesador moderno se encuentra en una etapa de refinamiento selectivo. La segmentación, la predicción, la ejecución fuera de orden y la especialización han sido decisivas para sostener el rendimiento, pero sus costos obligan a repensar el diseño bajo criterios de energía, área, verificabilidad y adecuación de carga. Las arquitecturas futuras no serán simplemente más rápidas por tener más hardware, sino más inteligentes en la distribución del trabajo, más conscientes del movimiento de datos, más adaptativas en el consumo y más especializadas sin perder compatibilidad. La tensión entre eficiencia energética y rendimiento no es un obstáculo externo al diseño; es el principio organizador de la microarquitectura contemporánea.

III. CONCLUSIONES

el microprocesador moderno debe comprenderse como una arquitectura dinámica de coordinación interna, no como una unidad secuencial simple dedicada únicamente a ejecutar instrucciones una tras otra. Su evolución ha estado marcada por la necesidad de sostener el rendimiento frente a límites físicos, energéticos y organizacionales que impiden depender exclusivamente del aumento de frecuencia. La segmentación instruccional, el paralelismo a nivel de instrucción, la predicción de saltos, la ejecución fuera de orden, las extensiones vectoriales, la microprogramación y la especialización funcional constituyen respuestas progresivas a un mismo problema: cómo transformar una secuencia lógica de instrucciones en una ejecución material eficiente, correcta y energéticamente viable.

El análisis del ciclo de instrucción y del pipelining permitió evidenciar que el rendimiento no surge únicamente de acelerar cada instrucción individual, sino de organizar el tiempo interno del procesador para que varias instrucciones avancen simultáneamente por etapas distintas. La división clásica en búsqueda, decodificación, ejecución, acceso a memoria y escritura de resultados ofrece una representación clara del flujo básico del procesamiento, pero también muestra sus límites. La segmentación incrementa el throughput, aunque introduce hazards estructurales, de datos y de control que obligan a incorporar forwarding, interlocks, stalls y flushes. Por ello, el pipeline no debe entenderse como una simple técnica de aceleración, sino como una arquitectura temporal que exige control preciso para preservar la corrección del programa.

La comparación entre RISC y CISC permitió reconocer que las categorías tradicionales de la arquitectura de instrucciones siguen siendo útiles, pero solo si se interpretan desde una mirada contemporánea. RISC aportó regularidad, simplicidad de codificación, facilidad de segmentación y eficiencia en sistemas embebidos y de bajo consumo. CISC aportó densidad de código, compatibilidad histórica y riqueza semántica. Sin embargo, los procesadores modernos han difuminado esas fronteras mediante traducción interna a micro-operaciones, extensiones SIMD, decodificadores avanzados, cachés de microinstrucciones, control energético y microarquitecturas híbridas. La pregunta central ya no es

qué paradigma es superior en abstracto, sino qué arquitectura resulta más pertinente para una carga, un ecosistema de software, una restricción energética y un contexto de aplicación determinados.

El estudio de las unidades de control microprogramadas permitió comprender que entre la ISA visible para el programador y la actividad física del hardware existe una capa de mediación. Las instrucciones no se convierten de manera automática en resultados; requieren señales de control, rutas internas, movimientos entre registros, operaciones de la ALU, accesos a memoria y actualizaciones de estado. La microprogramación hace visible esa traducción y muestra que el procesador puede ser analizado como una jerarquía de significados: el programa contiene instrucciones, las instrucciones pueden descomponerse en microoperaciones, y estas se materializan en señales que coordinan el datapath. Esta comprensión resulta fundamental tanto para la enseñanza universitaria como para el diseño de procesadores, simuladores, extensiones ISA y arquitecturas específicas de aplicación.

La predicción de saltos y la ejecución fuera de orden evidenciaron el grado de sofisticación alcanzado por la microarquitectura moderna. Para sostener pipelines profundos y unidades funcionales múltiples, el procesador debe anticipar rutas de control, especular sobre el futuro del programa, renombrar registros, seleccionar instrucciones listas, recuperar estados ante errores y comprometer resultados en orden arquitectónico. Estas técnicas permiten extraer paralelismo de programas aparentemente secuenciales, pero también incrementan consumo energético, complejidad de verificación, variabilidad temporal y costo de recuperación. La CPU contemporánea mantiene una apariencia externa secuencial y precisa, mientras internamente opera como una máquina dinámica, especulativa y altamente paralela.

La discusión crítica permitió establecer que el desafío principal de las microarquitecturas actuales no consiste solo en alcanzar mayor rendimiento, sino en producir trabajo útil con un costo energético razonable. Cada técnica de aceleración tiene un precio: los pipelines profundos aumentan penalizaciones por control; la predicción consume energía y puede fallar; la ejecución fuera de orden requiere estructuras complejas; las ventanas amplias presentan rendimientos

decrecientes; las unidades especializadas reducen flexibilidad; y los aceleradores dependen de datos correctamente ubicados y transferidos. En este sentido, la eficiencia energética no debe entenderse como una restricción externa al rendimiento, sino como una condición interna del diseño arquitectónico.

Finalmente, el refinamiento del hardware para soportar inteligencia artificial, computación científica, procesamiento digital de señales y sistemas embebidos exige avanzar hacia microarquitecturas equilibradas, heterogéneas y conscientes de la carga. La CPU seguirá siendo fundamental como unidad de control general, coordinación y ejecución flexible, pero su papel se complementará con SIMD, GPU, DSP, NPU, FPGA, extensiones ISA y aceleradores específicos. El futuro del microprocesador no dependerá únicamente de núcleos más complejos, sino de la integración inteligente entre propósito general y especialización, entre rendimiento y energía, entre compatibilidad y apertura, entre paralelismo implícito y paralelismo explícito. Así, el microprocesador moderno se proyecta como una arquitectura de equilibrio, capaz de sostener la computación científica y la inteligencia artificial solo si articula correctamente segmentación, control, predicción, memoria, energía y especialización funcional.

CAPITULO IV: PARADIGMAS EMERGENTES: ENTRADA/SALIDA, DMA Y VIRTUALIZACIÓN DE HARDWARE

Chapter IV: Emerging paradigms: input/output, dma, and hardware virtualization

CAPÍTULO IV: PARADIGMAS EMERGENTES: ENTRADA/SALIDA, DMA Y VIRTUALIZACIÓN DE HARDWARE

Chapter IV: Emerging paradigms: input/output, dma, and hardware virtualization

I. Introducción

La arquitectura y organización de computadores contemporánea ya no puede explicarse de manera suficiente desde una mirada centrada únicamente en el procesador, la memoria principal y la lógica interna de ejecución. Si bien estos elementos continúan siendo fundamentales, el rendimiento real de los sistemas modernos depende cada vez más de la forma en que los datos entran, salen, se transfieren, se aíslan, se virtualizan y se procesan en entornos heterogéneos, distribuidos y sensibles a la latencia. En este escenario, la entrada/salida deja de ser una periferia subordinada al procesador y se convierte en un eje estructural de la arquitectura computacional. Redes de cientos de gigabits, unidades NVMe, aceleradores GPU, dispositivos IoT, arquitecturas cloud, plataformas edge, sistemas virtualizados y centros de datos orientados a inteligencia artificial obligan a reconsiderar el papel de los mecanismos de E/S en el diseño de sistemas eficientes, seguros y escalables.

La interacción entre el procesador y el entorno externo siempre ha sido una condición indispensable para que la computación tenga sentido práctico. Un sistema computacional no solo ejecuta instrucciones internas, sino que recibe datos desde sensores, redes, almacenamiento, usuarios y dispositivos; los transforma mediante operaciones lógicas y aritméticas; y devuelve resultados hacia otros sistemas, actuadores, interfaces o repositorios persistentes. En las primeras arquitecturas, muchas operaciones de entrada/salida dependían de la intervención directa de la CPU, lo que implicaba un costo significativo en ciclos de procesamiento. A medida que los dispositivos crecieron en velocidad y volumen de transferencia, esta mediación directa se volvió insuficiente. El acceso directo a memoria, conocido como DMA, surgió precisamente como una estrategia para permitir que los dispositivos transfieran datos hacia o desde memoria sin requerir que el procesador participe en cada movimiento elemental.

Sin embargo, en los sistemas actuales, DMA ya no es únicamente una técnica de descarga de trabajo para la CPU; se ha convertido en un campo complejo de optimización arquitectónica, coherencia, localidad, seguridad y eficiencia energética.

La importancia contemporánea del DMA se evidencia en el crecimiento de las cargas intensivas en entrada/salida. En redes de alta velocidad, el tiempo disponible para procesar paquetes pequeños puede ser menor que la latencia de acceso a memoria principal, lo que obliga a utilizar mecanismos como Direct Cache Access y Data Direct I/O para ubicar datos de E/S directamente en la caché de último nivel. Farshin, Roozbeh, Maguire y Kostić (2020) muestran que, en redes de 100 y 200 Gbps, la gestión de caché para datos de E/S se vuelve decisiva, porque el acceso a memoria principal puede impedir que hardware convencional procese paquetes dentro de los márgenes temporales requeridos. Esta observación permite comprender que el problema de la E/S ya no se limita a mover datos, sino a decidir en qué nivel de la jerarquía deben ubicarse para que el procesamiento posterior no quede dominado por latencia, congestión o fallos de caché.

La relación entre DMA y localidad se vuelve todavía más crítica en servidores multisocket. En arquitecturas NUMA, cada CPU posee memoria local y accede a memoria remota mediante interconexiones internas con mayor latencia y menor ancho de banda. Cuando un dispositivo de E/S realiza DMA hacia memoria asociada a un nodo remoto, aparece el problema del DMA no uniforme, o NUDMA. Smolyar et al. (2020) explican que los dispositivos conectados a través de PCIe pueden experimentar diferencias de rendimiento según el nodo de memoria hacia el que transfieren datos, porque las operaciones remotas deben atravesar interconexiones entre CPU y, por tanto, reproducen efectos similares a NUMA en el plano de la E/S. Esto revela que la arquitectura de entrada/salida debe ser consciente de la topología física del sistema. No basta con que un dispositivo pueda acceder a memoria; importa desde dónde accede, hacia qué nodo transfiere, qué ruta atraviesa y qué impacto produce sobre cachés, memoria y aplicaciones.

En los sistemas de almacenamiento, el desafío es igualmente profundo. Las unidades NVMe han reducido drásticamente la latencia y aumentado el paralelismo de las operaciones de E/S, pero también han expuesto la ineficiencia de pilas de software tradicionales diseñadas para dispositivos más lentos. Zhu, Wang, Xiao y Qin (2023) sostienen que, aunque los SSD NVMe ofrecen alto rendimiento, el largo recorrido de la pila de E/S del sistema operativo puede degradar su potencial, especialmente por cambios de contexto, interrupciones, operaciones de fijación de memoria y sobrecostos vinculados al acceso desde espacio de usuario. Esta situación explica la expansión de marcos como SPDK, user-level I/O, polling, colas lock-free, memoria fijada y mecanismos adaptativos de DMA. El objetivo no es solo acelerar el dispositivo, sino reducir el costo de la mediación software entre aplicación, sistema operativo, controlador, memoria y almacenamiento.

La evolución hacia almacenamiento desagregado y NVMe-over-Fabrics amplía todavía más esta problemática. En centros de datos y nubes HPC, las aplicaciones pueden ejecutarse en máquinas virtuales o contenedores mientras acceden a almacenamiento remoto a través de redes Ethernet, RDMA o fabrics especializados. Kashyap y Lu (2022) proponen NVMe-over-Adaptive-Fabric para reducir los costos de latencia y bajo ancho de banda asociados al acceso remoto por TCP/IP, incorporando conciencia de localidad y selección adaptativa entre memoria compartida y canales de red. Esta propuesta evidencia que la E/S moderna no puede reducirse a un dispositivo local conectado a una placa base. La E/S se despliega ahora en una topología distribuida donde el almacenamiento puede estar separado del cómputo, el canal de comunicación puede variar según ubicación y el rendimiento depende de seleccionar dinámicamente la ruta más adecuada para cada transferencia.

Los aceleradores profundizan esta transformación. En plataformas de Big Data, inteligencia artificial y computación científica, los datos pueden necesitar desplazarse entre almacenamiento NVMe, memoria del host y memoria de GPU. El modelo tradicional, donde los datos pasan del almacenamiento a la memoria de CPU y luego a la GPU, introduce copias intermedias que limitan el throughput. Bayati, Leeser y Mi (2020) muestran que el acceso directo de GPU a memoria no volátil mediante GPUDirect y DMA peer-to-peer puede reducir el tránsito por

memoria del host y mejorar el rendimiento de aplicaciones de procesamiento de datos en Spark. Esta línea de investigación confirma que el rendimiento de sistemas acelerados no depende únicamente de la potencia de la GPU, sino de la capacidad de alimentar sus unidades de cómputo con datos sin imponer trayectorias redundantes por la CPU.

Junto con DMA, las interrupciones y el polling constituyen mecanismos esenciales para coordinar la relación entre CPU y dispositivos. Las interrupciones permiten que el dispositivo notifique eventos al procesador solo cuando requiere atención, lo que evita que la CPU permanezca verificando continuamente el estado del periférico. No obstante, en cargas de E/S muy intensivas, el costo de atender interrupciones frecuentes, cambiar de contexto y recorrer la pila del sistema operativo puede resultar prohibitivo. El polling, en cambio, permite que la CPU consulte de manera activa colas o registros de dispositivo, reduciendo latencia y evitando sobrecostos de interrupción, aunque a cambio consume ciclos de CPU incluso cuando no hay trabajo disponible. En este sentido, la elección entre interrupciones y sondeo no es meramente técnica, sino arquitectónica: depende de la intensidad de la carga, los requisitos de latencia, el consumo energético aceptable, el número de núcleos disponibles y la previsibilidad requerida por la aplicación.

La virtualización introduce una capa adicional de complejidad sobre estos mecanismos. En entornos cloud, industriales, embebidos o de alta disponibilidad, varios sistemas operativos y aplicaciones pueden compartir el mismo hardware físico mediante máquinas virtuales administradas por un hipervisor. Esta abstracción permite consolidación, aislamiento, migración, flexibilidad y mejor utilización de recursos, pero también genera sobrecostos en CPU, memoria, traducción de direcciones, interrupciones y dispositivos de E/S. Sá et al. (2023) explican que la virtualización se utiliza desde cloud computing hasta sistemas embebidos, y que las arquitecturas modernas han incorporado soporte de hardware para reducir su sobrecosto, como ocurre en RISC-V mediante la extensión Hypervisor y mejoras microarquitectónicas en TLB y MMU anidada. Esto demuestra que la virtualización ya no es solo una técnica de software, sino una propiedad arquitectónica que exige cooperación entre ISA, microarquitectura, memoria y sistema operativo.

La seguridad de la virtualización es igualmente decisiva. Un hipervisor ocupa una posición privilegiada porque administra el acceso de las máquinas virtuales a CPU, memoria, almacenamiento y dispositivos. Si esta capa es comprometida, el atacante podría afectar múltiples máquinas virtuales y acceder a recursos compartidos. Aalam, Kumar y Gour (2021) describen el hipervisor como una capa de aislamiento y control que administra recursos físicos para múltiples máquinas virtuales, pero también advierten que su posición privilegiada incrementa la superficie de ataque, especialmente porque tiene visibilidad sobre registros, memoria e I/O de las máquinas hospedadas. Esta tensión entre abstracción, seguridad y rendimiento atraviesa todo el capítulo: cuanto más flexible y compartido es el sistema, más importante se vuelve garantizar aislamiento, integridad y control sobre la mediación de recursos.

En sistemas embebidos y críticos, la virtualización adopta un sentido particular. No siempre se busca elasticidad dinámica como en la nube, sino aislamiento temporal y espacial, reducción del TCB, certificabilidad y comportamiento verificable. Nordholz (2020) propone un hipervisor embebido con mínima dinámica en tiempo de ejecución, desplazando decisiones de configuración hacia tiempo de compilación para hacer más tratable la verificación simbólica del binario final. Esta aproximación resulta relevante porque muestra que la virtualización no es un paradigma homogéneo. En la nube puede priorizar consolidación y elasticidad; en sistemas embebidos puede priorizar seguridad, previsibilidad y verificabilidad; en HPC puede priorizar baja sobrecarga; en edge computing puede buscar aislamiento con recursos limitados.

La arquitectura orientada a la nube amplía la escala de este análisis. El cloud computing transformó la infraestructura computacional al convertir cómputo, almacenamiento, redes y servicios en recursos bajo demanda, escalables y administrados mediante virtualización, automatización y modelos de servicio. Sehgal, Bhatt y Acken (2023) presentan la nube como un cambio arquitectónico apoyado en virtualización, multi-tenancy, provisión bajo demanda, orientación a servicios, elasticidad y acuerdos de nivel de servicio, articulando principios técnicos y económicos de la computación como utilidad. Bajo esta lógica, la organización computacional ya no reside únicamente dentro de una máquina,

sino en la coordinación de centros de datos, redes, hipervisores, contenedores, servicios, políticas de seguridad y mecanismos de escalamiento.

No obstante, el modelo cloud centralizado presenta límites cuando las aplicaciones demandan baja latencia, continuidad operativa, privacidad local o respuesta en tiempo real. Edge computing surge como respuesta al desplazar procesamiento y almacenamiento hacia nodos cercanos a los usuarios, sensores o dispositivos industriales. Harjula, Artemenko y Forsström (2022) sostienen que el edge computing permite reducir latencia y vulnerabilidad frente a problemas de red al ubicar recursos computacionales cerca de los dispositivos finales, lo cual resulta especialmente relevante en Industrial IoT, 5G, automatización y servicios con requisitos estrictos de disponibilidad y predictibilidad. Este desplazamiento modifica la organización de la E/S: los datos ya no siempre viajan hacia un centro remoto para ser procesados, sino que pueden filtrarse, analizarse, inferirse o actuar localmente antes de enviarse a la nube.

Los sistemas edge, fog y mist no sustituyen por completo a la nube, sino que introducen una continuidad arquitectónica entre dispositivos, nodos periféricos, redes de acceso, centros regionales y data centers. En este continuo, la ubicación del procesamiento se decide según latencia, ancho de banda, privacidad, costo, energía y criticidad. Ferreira et al. (2022) muestran esta lógica en sistemas de gestión energética del hogar, al proponer un middleware basado en microservicios ejecutado en hardware de bajo costo cerca del usuario, capaz de reducir dependencia de conexiones externas, disminuir tiempo de respuesta y operar bajo restricciones de procesamiento y almacenamiento. Este tipo de solución evidencia que el edge no es solo una versión pequeña de la nube, sino una organización distinta, donde las restricciones del hardware, la proximidad de los datos y la continuidad del servicio son elementos centrales del diseño.

El avance de arquitecturas distribuidas también se relaciona con nuevos paradigmas de computación paralela y emergente. Dai, Hossain y Wang (2025) describen la evolución de sistemas paralelos y distribuidos hacia modelos heterogéneos, cloud-native, serverless, blockchain, inteligencia artificial distribuida, computación neuromórfica y óptica, destacando desafíos de

escalabilidad, seguridad, interoperabilidad, tolerancia a fallos e integración de recursos diversos. Esta visión permite situar el capítulo dentro de una transición más amplia: los sistemas computacionales se organizan cada vez más como redes de recursos heterogéneos, donde CPU, GPU, NPU, almacenamiento, red, hipervisores, contenedores y nodos edge deben cooperar para responder a cargas de trabajo cambiantes.

La sostenibilidad aparece como una dimensión transversal de esta transformación. Los centros de datos dedicados a inteligencia artificial consumen cantidades crecientes de energía, y las arquitecturas tradicionales basadas en CPU, GPU y TPU enfrentan límites para sostener la demanda de cómputo de modelos cada vez más grandes. Vogginger et al. (2024) plantean que el hardware neuromórfico, inspirado en el procesamiento cerebral y basado en eventos, puede contribuir a una inteligencia artificial más eficiente energéticamente, aunque aún enfrenta desafíos de integración con software, algoritmos y entornos de data center. En esta línea, la organización computacional futura no solo deberá ser rápida y escalable, sino también energéticamente responsable, compatible con nuevas formas de hardware y capaz de integrar aceleradores no convencionales dentro de infraestructuras operativas reales.

El presente capítulo se desarrolla a partir de cuatro ejes. En primer lugar, se examina la gestión de periféricos y el acceso directo a memoria, considerando DMA tradicional, DDIO, NUDMA, DMA a nivel de usuario, NVMe, NVMe-oF, GPUDirect Storage y optimización de rutas de datos. En segundo lugar, se analiza la relación entre interrupciones de hardware y polling, mostrando cómo la búsqueda de baja latencia puede entrar en tensión con consumo de CPU, eficiencia energética y escalabilidad. En tercer lugar, se estudia el impacto de la virtualización en la abstracción de recursos físicos, incorporando hipervisores, seguridad, RISC-V, TLB, MMU anidada, verificación simbólica y sistemas embebidos. En cuarto lugar, se abordan las arquitecturas orientadas a la nube, incluyendo cloud, edge, fog, mist, microservicios, cloud-native, sistemas distribuidos y hardware neuromórfico.

La discusión crítica del capítulo problematizará el desplazamiento del procesamiento hacia la periferia y la necesidad de nuevas arquitecturas de E/S.

Este debate no se limitará a oponer cloud y edge, sino que analizará cómo la baja latencia, la privacidad, la disponibilidad, el consumo energético, la virtualización y el movimiento de datos obligan a repensar la organización computacional. En la próxima década, la eficiencia no dependerá únicamente de procesadores más rápidos, sino de sistemas capaces de reducir copias, evitar trayectorias innecesarias, virtualizar recursos con bajo sobrecosto, proteger datos en tránsito, seleccionar dinámicamente rutas de E/S y ubicar procesamiento cerca del lugar donde los datos adquieren valor. En este sentido, entrada/salida, DMA y virtualización de hardware constituyen paradigmas emergentes porque redefinen la frontera entre máquina, red, nube, periferia y aplicación.

II. DESARROLLO

Gestión de periféricos y acceso directo a memoria (DMA)

La gestión de periféricos constituye una dimensión esencial de la organización computacional porque define la manera en que el sistema interactúa con dispositivos externos, almacenamiento, redes, sensores, aceleradores y controladores especializados. A diferencia de la ejecución interna del procesador, donde el flujo de instrucciones se organiza en torno a registros, ALU, cachés y unidades de control, la entrada/salida introduce una relación con componentes de velocidad, latencia, protocolo y comportamiento heterogéneos. Un periférico puede ser un dispositivo relativamente lento, como un teclado o un sensor de baja frecuencia, pero también puede ser una tarjeta de red de 200 o 400 Gbps, una unidad NVMe de alta concurrencia, una GPU con memoria propia, un controlador de almacenamiento remoto o un dispositivo IoT conectado al borde de la red. Esta diversidad obliga a que la arquitectura de E/S no se limite a un único mecanismo de comunicación, sino que combine registros de control, colas, descriptores, interrupciones, polling, DMA, buses, coherencia de memoria y políticas de seguridad.

En los sistemas clásicos, una forma simple de entrada/salida consistía en que la CPU participara directamente en la transferencia de datos entre el dispositivo y la memoria. Esta estrategia resultaba aceptable cuando los dispositivos eran lentos y los volúmenes de datos pequeños, pero se volvió ineficiente conforme

aumentaron la velocidad de redes, almacenamiento y periféricos. Si la CPU debe intervenir en cada palabra o bloque transferido, una parte significativa de su tiempo se consume en mover datos en lugar de ejecutar lógica de aplicación. El acceso directo a memoria respondió a esta limitación al permitir que un dispositivo transfiera datos hacia o desde la memoria principal sin que la CPU participe en cada movimiento elemental. En este modelo, el procesador configura la operación, entrega direcciones o descriptores, y el controlador DMA ejecuta la transferencia de manera autónoma. La CPU queda liberada para otras tareas, aunque debe coordinar inicio, finalización, consistencia y manejo de errores.

El DMA transformó la relación entre procesador y periféricos porque desplazó la E/S desde una actividad controlada ciclo a ciclo por la CPU hacia una actividad delegada al dispositivo o al controlador. Sin embargo, esta delegación no elimina la complejidad. Para que una operación DMA sea correcta, el sistema debe garantizar que las direcciones sean válidas, que la memoria esté disponible, que no se produzcan conflictos de coherencia, que los buffers no sean modificados indebidamente durante la transferencia y que el dispositivo no acceda a regiones no autorizadas. En sistemas modernos, esta coordinación involucra controladores, IOMMU, páginas fijadas, colas de descriptores, políticas de caché y mecanismos de protección. El DMA, por tanto, no es únicamente una vía rápida de transferencia; es una relación de confianza regulada entre dispositivo y memoria.

La evolución de los dispositivos de almacenamiento NVMe evidencia por qué el DMA se volvió crítico. Los SSD NVMe ofrecen paralelismo, baja latencia y alto throughput, pero su aprovechamiento puede quedar limitado por la pila de software del sistema operativo. Cuando una aplicación emite una solicitud de E/S, el recorrido tradicional puede incluir llamadas al sistema, cambios entre espacio de usuario y kernel, planificación de E/S, manejo de interrupciones, asignación de buffers, traducción de direcciones y sincronización. Zhu, Wang, Xiao y Qin (2023) señalan que, aunque NVMe SSD proporciona alto rendimiento, la longitud de la pila de E/S puede convertirse en un cuello de botella, especialmente por cambios de contexto, interrupciones y operaciones repetidas de fijación de memoria para DMA. Esta observación permite comprender que el

problema ya no reside solo en la velocidad del dispositivo, sino en la eficiencia del camino completo entre aplicación, memoria, controlador y almacenamiento.

Las estrategias de user-level I/O buscan precisamente reducir la mediación del kernel en el camino crítico de E/S. Marcos como SPDK trasladan parte del controlador y de la gestión de colas hacia espacio de usuario, utilizan polling, colas lock-free y memoria fijada para reducir latencia y aumentar paralelismo. No obstante, estos enfoques introducen nuevos problemas. Si cada solicitud requiere fijar memoria o administrar buffers de forma costosa, el beneficio de evitar el kernel puede reducirse. Zhu et al. (2023) proponen uDMA como un mecanismo de DMA a nivel de usuario que reutiliza bloques de memoria fijada mediante un pinned memory pool y emplea listas scatter/gather para conectar bloques discretos, con el objetivo de amortizar la latencia por solicitud y adaptarse a tamaños variables de E/S. Esta solución muestra que la eficiencia del DMA depende tanto de evitar copias como de administrar inteligentemente la memoria asociada a la transferencia.

La fijación de memoria es una condición importante para DMA porque el dispositivo necesita direcciones estables durante la operación. En sistemas con memoria virtual, una página puede moverse, intercambiarse o cambiar su correspondencia física si no se fija adecuadamente. Para la CPU, la abstracción de memoria virtual facilita protección y flexibilidad; para un dispositivo que ejecuta DMA, la estabilidad de las direcciones y la traducción correcta son condiciones esenciales. Cuando las operaciones de fijación se repiten con demasiada frecuencia, generan sobrecarga. Por ello, la reutilización de buffers fijados y su organización mediante scatter/gather permite que el sistema aproveche memoria previamente preparada para DMA sin incurrir en el costo de inicialización en cada solicitud. Esta idea revela que el rendimiento de E/S no depende solo del bus o del dispositivo, sino de la administración de memoria que hace posible la transferencia.

Las listas scatter/gather permiten que una transferencia DMA utilice múltiples regiones de memoria no contiguas como si formaran una operación lógica coherente. Esta capacidad es importante porque la memoria física disponible puede estar fragmentada y porque las aplicaciones no siempre trabajan con

buffers contiguos. En lugar de copiar datos hacia una región continua antes de transferirlos, el sistema describe al dispositivo una lista de segmentos. Así se reducen copias, se mejora el uso de memoria y se incrementa la flexibilidad de la operación. En almacenamiento NVMe, redes de alta velocidad y sistemas user-level, esta técnica contribuye a que el DMA sea más adaptable a cargas reales, donde los tamaños, patrones y ubicaciones de datos varían continuamente.

El DMA tradicional suele transferir datos entre dispositivo y memoria principal, pero en redes de muy alta velocidad este modelo puede resultar insuficiente. Cuando un paquete llega desde una tarjeta de red, escribirlo primero en DRAM y luego cargarlo en caché para procesarlo introduce latencias y consumo de ancho de banda que pueden ser inaceptables. Direct Cache Access y, en particular, Intel Data Direct I/O intentan reducir este costo al permitir que los dispositivos coloquen datos directamente en la caché de último nivel. Farshin, Roozbeh, Maguire y Kostić (2020) explican que DDIO permite que una NIC transfiera datos directamente hacia o desde la LLC en lugar de usar memoria principal como destino primario, lo que reduce accesos a DRAM y puede mejorar latencia y throughput en aplicaciones de red intensivas. En este caso, la E/S deja de ser una comunicación dispositivo-memoria y se convierte en una interacción dispositivo-caché.

El valor de DDIO se entiende mejor al considerar la escala temporal de redes de 100 o 200 Gbps. En esas velocidades, el intervalo entre paquetes pequeños puede ser de pocos nanosegundos, mientras que un acceso a memoria principal puede tardar mucho más. Si el procesador debe recuperar cada paquete desde DRAM, el sistema queda rápidamente limitado por latencia y ancho de banda de memoria. Al ubicar los datos en la LLC, DDIO aproxima el dato de E/S al lugar donde será procesado. No obstante, esta optimización también introduce riesgos. La caché es un recurso limitado y compartido. Si demasiados datos de E/S ingresan a la LLC, pueden desplazar datos útiles de las aplicaciones o producir variabilidad en latencias de cola. Farshin et al. (2020) muestran que el ajuste de DDIO puede reducir latencia en funciones de red, pero también advierten que en tráfico de 200 Gbps puede incrementar latencias de cola si no se gestiona selectivamente la inyección de datos en caché.

La investigación de Sharan, Vutukuru y Panda (2023) refuerza esta problemática al proponer DDIOSim como un simulador microarquitectónico capaz de modelar el procesamiento de paquetes con DDIO, incluyendo interacción entre NIC, CPU, cachés y jerarquía de memoria. Su aporte es relevante porque muestra que las decisiones de E/S deben estudiarse en relación con la microarquitectura completa, no como un subsistema separado. DDIO interactúa con políticas de caché, prefetchers, planificadores de DRAM, aplicaciones de red y comportamiento del kernel. Esta interacción confirma que el diseño de E/S moderna exige herramientas que permitan observar efectos cruzados entre hardware y software, especialmente cuando se busca rendimiento estable bajo cargas intensivas.

La no uniformidad del acceso es otro problema central en la gestión de DMA. En servidores multisocket, la topología física condiciona el costo de transferir datos entre dispositivo, CPU y memoria. Un dispositivo PCIe suele estar conectado a un socket específico; si realiza DMA hacia memoria local a ese socket, la ruta es más eficiente; si accede a memoria de otro nodo, la transferencia atraviesa interconexiones internas, generando mayor latencia y menor throughput. Smolyar et al. (2020) denominan este fenómeno NUDMA y sostienen que su impacto se asemeja al NUMA tradicional, aunque proviene de la ubicación del dispositivo dentro de la topología de E/S. Esta observación obliga a ampliar el concepto de localidad: ya no basta ubicar los datos cerca del núcleo que los procesará; también importa que el dispositivo que los produce o consume esté conectado al nodo adecuado.

IOctopus propone una forma de replantear el problema al conectar conceptualmente un dispositivo a varios sockets mediante múltiples funciones PCIe que aparecen como una sola entidad lógica ante el sistema. La intuición es que NUDMA no es inevitable del mismo modo que NUMA, porque los dispositivos de E/S son externos a la topología CPU-memoria y podrían dirigir sus transferencias hacia el endpoint más adecuado. Smolyar et al. (2020) muestran que esta arquitectura puede mejorar throughput y latencia al evitar que las operaciones DMA remotas se comporten como una penalización estructural permanente. El aporte conceptual es significativo: la arquitectura del dispositivo

puede intervenir activamente en la localidad de las transferencias, en lugar de dejar que el sistema operativo compense un diseño físico subóptimo.

En almacenamiento remoto, el problema se desplaza desde la topología interna del servidor hacia la topología de red y virtualización. NVMe-over-Fabrics permite acceder a dispositivos NVMe remotos mediante transportes como TCP, RDMA, InfiniBand o Ethernet. Esta estrategia facilita desagregación de almacenamiento, mayor utilización de recursos y escalabilidad en centros de datos, pero introduce latencia de red y complejidad de transporte. Kashyap y Lu (2022) sostienen que, en nubes HPC, las aplicaciones que se ejecutan en contenedores o máquinas virtuales suelen transformar sus accesos de almacenamiento en E/S remota, por lo que el rendimiento de la red y del transporte se vuelve decisivo para cargas intensivas. En este contexto, el DMA ya no ocurre solo dentro de una máquina; forma parte de un trayecto distribuido que puede incluir VMs, servicios de almacenamiento, memoria compartida, TCP/IP y dispositivos NVMe remotos.

NVMe-oAF, como propuesta adaptativa, introduce una idea útil para el diseño contemporáneo: no todas las transferencias deben usar el mismo camino. Si la aplicación y el servicio de almacenamiento se encuentran en el mismo nodo, un canal de memoria compartida puede evitar la red y reducir latencia; si se encuentran en nodos distintos, se requiere transporte TCP optimizado u otra forma de comunicación. La arquitectura adaptativa decide según localidad y condiciones del entorno. Kashyap y Lu (2022) reportan mejoras relevantes en ancho de banda y latencia al combinar memoria compartida, optimización de TCP, detección de tamaño de chunks y selección adaptativa del canal. Esta estrategia confirma que la E/S moderna debe ser consciente de contexto: la ruta óptima depende de dónde están los procesos, dónde residen los datos, qué canal está disponible y qué tipo de carga se ejecuta.

La relación entre DMA y aceleradores se observa con especial claridad en GPU Direct Storage. En el modelo tradicional, los datos viajan desde almacenamiento hacia memoria del host y luego desde allí hacia memoria de GPU. En cargas de Big Data, aprendizaje automático o procesamiento masivo, esta doble transferencia puede limitar el rendimiento de la GPU, porque sus unidades de

cómputo esperan datos que atraviesan una ruta innecesariamente larga. Bayati, Leeser y Mi (2020) integran Spark-GPU con acceso directo entre GPU y NVMe mediante DMA peer-to-peer, mostrando que al evitar la memoria del host se mejora el throughput de transferencia y el rendimiento de extremo a extremo. Esta propuesta ilustra una tendencia mayor: los datos deben moverse de forma directa entre los dispositivos que los almacenan y los que los procesan, sin pasar por intermediarios que no agregan valor computacional.

El acceso directo entre GPU y almacenamiento también modifica el papel de la CPU. La CPU deja de ser el punto obligado de paso para todos los datos y se convierte en coordinadora, planificadora o administradora del flujo. Esta transformación es coherente con la evolución hacia sistemas heterogéneos, donde CPU, GPU, DPU, NIC, SSD y aceleradores especializados cooperan mediante interconexiones y protocolos de memoria. La dificultad consiste en mantener seguridad, coherencia y control cuando se habilitan rutas directas entre dispositivos. Si un acelerador puede acceder a almacenamiento o memoria sin atravesar el camino tradicional del kernel, el sistema debe garantizar que solo acceda a regiones autorizadas, que las transferencias sean consistentes y que los errores puedan ser detectados y recuperados.

En escenarios de edge computing, el DMA adquiere otra dimensión. Los dispositivos periféricos pueden incluir cámaras, sensores espectrales, controladores industriales, actuadores o módulos de inferencia local. Zhang et al. (2024) proponen ISMSFuse para reconocimiento multimodal de enfermedad bacteriana en arroz, combinando una red MobileNetV2 ligera con SVM lineal y desplegando el modelo en Raspberry Pi para verificar su factibilidad en edge computing. Aunque el estudio se centra en agricultura inteligente, su relevancia arquitectónica radica en mostrar que el edge exige algoritmos y flujos de datos compatibles con hardware limitado, baja latencia y adquisición local de datos. En estos entornos, la gestión de E/S no es una función secundaria: determina si el sistema puede capturar, procesar y responder localmente sin depender de la nube.

La entrada/salida en el borde no siempre involucra dispositivos de máximo throughput, pero sí restricciones estrictas de energía, costo, conectividad y

tiempo de respuesta. Un sistema agrícola, industrial o energético puede requerir procesar datos de sensores en tiempo casi real con hardware de bajo costo. Si cada dato debe enviarse a la nube, se incrementan latencia, dependencia de red y costo operativo. Si se procesa localmente, la arquitectura debe optimizar adquisición, memoria, inferencia, almacenamiento temporal y comunicación selectiva. En este contexto, DMA, buses internos, interfaces de sensores, colas y buffers determinan la eficiencia del sistema tanto como el modelo de aprendizaje automático utilizado.

La gestión contemporánea de periféricos, entonces, puede entenderse como una arquitectura del movimiento de datos. Los dispositivos no son apéndices pasivos; producen, consumen, transforman o aceleran información. Las NIC introducen paquetes a velocidades que pueden saturar memoria; los SSD NVMe exigen pilas de E/S ligeras; las GPU demandan trayectorias directas de datos; los sistemas NUMA requieren conciencia de localidad; las VMs y contenedores agregan capas de abstracción; y el edge impone restricciones de latencia y recursos. En todos estos casos, el DMA se mantiene como técnica fundamental, pero deja de ser un mecanismo único y uniforme. Se fragmenta en variantes: DMA tradicional, DDIO, user-level DMA, peer-to-peer DMA, NUDMA-aware DMA, NVMe-oF y acceso directo entre almacenamiento y aceleradores.

Esta diversidad obliga a pensar la E/S como un problema de co-diseño entre hardware y software. El hardware debe ofrecer rutas rápidas, protección, colas eficientes, coherencia y soporte de topología; el software debe asignar buffers, fijar memoria, elegir canales, ubicar tareas, administrar interrupciones o polling, y evitar copias innecesarias. Una optimización puramente hardware puede fallar si el software no coloca los datos donde corresponde; una optimización software puede quedar limitada por un diseño físico que obliga a rutas remotas o cuellos de botella. Por ello, los sistemas modernos requieren que controladores, sistemas operativos, hipervisores, runtimes y aplicaciones sean conscientes de las propiedades de la E/S.

En síntesis, la gestión de periféricos y el acceso directo a memoria constituyen un componente decisivo de las arquitecturas emergentes. El DMA nació como una técnica para liberar a la CPU del movimiento repetitivo de datos, pero hoy se ha

convertido en un campo donde convergen rendimiento, localidad, memoria virtual, caché, redes, almacenamiento, aceleradores, virtualización y edge computing. Su evolución muestra que el problema central de la computación contemporánea no es únicamente calcular más rápido, sino mover menos, mover mejor, mover con seguridad y mover por la ruta arquitectónicamente más adecuada.

Interrupciones de hardware vs. sondeo (polling)

La relación entre interrupciones de hardware y sondeo constituye uno de los dilemas clásicos de la organización de entrada/salida, pero en las arquitecturas contemporáneas adquiere una relevancia renovada debido al crecimiento de redes de alta velocidad, almacenamiento NVMe, virtualización, contenedores, sistemas cloud y procesamiento en tiempo real. Ambos mecanismos buscan resolver una misma necesidad: coordinar el momento en que la CPU debe atender un evento producido por un dispositivo. Sin embargo, lo hacen desde lógicas opuestas. La interrupción permite que el periférico notifique al procesador cuando requiere atención, evitando que la CPU consuma ciclos revisando continuamente su estado. El polling, en cambio, implica que el procesador consulte activamente una cola, registro o descriptor para detectar si existe trabajo pendiente. La primera estrategia privilegia ahorro de CPU y respuesta bajo demanda; la segunda privilegia latencia baja y control continuo sobre el flujo de E/S.

En sistemas tradicionales, las interrupciones fueron una solución eficiente porque los dispositivos eran relativamente lentos y los eventos de E/S no ocurrían con una frecuencia tan alta como para saturar al procesador. El dispositivo generaba una señal de interrupción, la CPU suspendía temporalmente su flujo de ejecución, guardaba el contexto necesario y transfería el control a una rutina de servicio. Una vez atendido el evento, el sistema retornaba a la tarea anterior. Este modelo permitió evitar el desperdicio de ciclos asociado al sondeo permanente. No obstante, la interrupción no es gratuita. Su atención implica cambios de contexto, invalidación o perturbación de caché, ejecución de código del kernel, sincronización y eventual despertar de procesos.

Cuando los eventos son esporádicos, ese costo es aceptable; cuando ocurren millones de veces por segundo, puede convertirse en un cuello de botella.

El polling surge como una alternativa especialmente atractiva en cargas intensivas de E/S, donde se espera que siempre haya trabajo disponible o donde la latencia de reacción debe reducirse al mínimo. En lugar de esperar una interrupción, un núcleo puede quedar dedicado a revisar continuamente una cola de completación, un anillo de descriptores o un canal de comunicación con el dispositivo. Esta estrategia elimina parte del costo de notificación y reduce variabilidad, porque la CPU detecta el evento de manera directa y predecible. Sin embargo, su costo principal es evidente: consume recursos de CPU incluso cuando no hay solicitudes pendientes. Por tanto, el polling mejora latencia a cambio de ocupar ciclos que podrían utilizarse para otras tareas. En centros de datos, HPC y almacenamiento NVMe, esta elección suele justificarse cuando la reducción de latencia y el aumento de throughput superan el costo de dedicar núcleos al sondeo.

El caso de SPDK permite comprender por qué el polling se volvió central en la optimización de almacenamiento NVMe. Al trasladar controladores al espacio de usuario, evitar llamadas frecuentes al sistema y emplear colas lock-free, SPDK reduce gran parte del recorrido clásico de la pila de E/S. Zhu, Wang, Xiao y Qin (2023) explican que SPDK utiliza polling, modo lock-free y DMA a nivel de usuario para ofrecer acceso altamente paralelo a SSD NVMe desde aplicaciones en espacio de usuario, aunque también advierten que el rendimiento puede degradarse si la gestión de memoria fijada introduce sobrecostos adicionales. Esta observación revela que el polling no actúa de forma aislada. Su eficacia depende de que las demás capas de la ruta de E/S, como buffers, memoria DMA, colas y asignación de núcleos, estén igualmente optimizadas.

En el procesamiento de paquetes de red, el dilema es similar. Una NIC de alta velocidad puede recibir paquetes con una frecuencia tan elevada que interrumpir al procesador por cada evento sería inviable. En estos escenarios, los sistemas suelen recurrir a técnicas híbridas, como interrupciones moderadas, coalescencia de interrupciones, polling adaptativo o núcleos dedicados a procesamiento de paquetes. Farshin, Roozbeh, Maguire y Kostić (2020)

muestran que, en redes de 100 y 200 Gbps, la latencia y la variabilidad no dependen únicamente de la velocidad de la NIC, sino también de la manera en que los datos ingresan a la jerarquía de caché mediante DDIO y de cómo el procesador administra la carga de paquetes. En este contexto, el polling puede reducir el costo de notificación, pero no resuelve por sí solo problemas de caché, localidad, congestión o memoria.

La coalescencia de interrupciones representa una solución intermedia. En lugar de generar una interrupción por cada evento, el dispositivo agrupa múltiples completaciones o paquetes antes de interrumpir a la CPU. Esto reduce la tasa de interrupciones y mejora eficiencia, pero introduce latencia adicional, porque algunos eventos esperan hasta que se alcance un umbral de tiempo o cantidad. Esta estrategia es útil cuando el objetivo es aumentar throughput y reducir sobrecarga del kernel, pero puede afectar aplicaciones sensibles a latencia. El polling adaptativo, por su parte, intenta combinar lo mejor de ambos enfoques: utilizar sondeo cuando la carga es alta y retornar a interrupciones cuando la carga disminuye. La dificultad consiste en definir umbrales adecuados y evitar oscilaciones que produzcan inestabilidad.

En almacenamiento remoto y NVMe-over-Fabrics, esta tensión se amplía porque la ruta de E/S atraviesa redes, máquinas virtuales, contenedores y servicios de almacenamiento. El protocolo puede operar sobre TCP/IP, RDMA, InfiniBand o Ethernet, y cada transporte introduce diferentes costos de CPU, latencia, administración y compatibilidad. Kashyap y Lu (2022) plantean que el acceso NVMe remoto en nubes HPC puede verse limitado por la ruta TCP/IP, por lo que proponen una arquitectura adaptativa que selecciona entre memoria compartida y TCP optimizado según localidad, incorporando busy polling, detección de tamaño de chunks y mecanismos de reducción de copias. Esta propuesta muestra que polling e interrupciones deben analizarse en relación con el canal de comunicación completo. Si el almacenamiento está dentro del mismo nodo, el polling sobre memoria compartida puede ser muy eficiente; si está remoto, el costo de red redefine la estrategia.

El sondeo también se relaciona con la predictibilidad temporal. En sistemas de baja latencia, el tiempo de reacción ante un evento no depende de cuándo el

dispositivo decide interrumpir, ni de cuándo el sistema operativo agenda la rutina correspondiente, sino del intervalo con que la CPU revisa la cola. Esta regularidad puede ser valiosa en redes financieras, procesamiento de paquetes, almacenamiento HPC o sistemas de control. Sin embargo, la predictibilidad del polling puede verse afectada si el núcleo comparte recursos con otras tareas, si existe contención de caché, si la topología NUMA ubica buffers lejos del núcleo que sondea o si la virtualización introduce capas adicionales. Por ello, el polling eficiente suele requerir afinidad de CPU, asignación cuidadosa de memoria, aislamiento de núcleos y control del entorno de ejecución.

El uso de polling en espacio de usuario ha cambiado la frontera entre sistema operativo y aplicación. Tradicionalmente, el kernel administraba dispositivos, interrupciones, colas, memoria y seguridad. Los marcos user-level trasladan parte de esa responsabilidad a bibliotecas y runtimes, reduciendo latencia, pero exigiendo mayor cuidado en protección, asignación de recursos y control de acceso. Esta transformación tiene ventajas claras en rendimiento, pero también implica que la aplicación o la biblioteca deben gestionar elementos que antes eran invisibles. El diseño arquitectónico se vuelve más explícito: el desarrollador debe considerar memoria fijada, colas, núcleos dedicados, tamaño de lote, alineación, canales de comunicación y afinidad de dispositivo.

La virtualización complica aún más la atención de eventos de E/S. En una máquina virtual, una interrupción física puede ser mediada por el hipervisor, traducida o inyectada como interrupción virtual al sistema invitado. Del mismo modo, una cola de E/S puede estar virtualizada, paravirtualizada o asignada directamente mediante passthrough. Cada alternativa introduce diferentes compromisos entre rendimiento, aislamiento y flexibilidad. Un dispositivo asignado directamente a una VM puede reducir sobrecosto, pero limita migración y aumenta exigencias de aislamiento mediante IOMMU. Una E/S emulada ofrece compatibilidad, pero suele ser más lenta. Una interfaz paravirtualizada reduce parte del costo, pero requiere controladores específicos. En este contexto, interrupciones y polling no son decisiones puramente locales de un dispositivo, sino mecanismos afectados por la política de virtualización.

El polling tiene además un costo energético relevante. Un núcleo que sondea continuamente puede permanecer activo aun cuando no existan eventos, impidiendo estados profundos de ahorro de energía. En sistemas donde el objetivo es mínima latencia, este costo puede asumirse. En sistemas móviles, edge, IoT o plataformas de bajo consumo, puede ser inaceptable. Por ello, los diseños modernos suelen requerir estrategias adaptativas que equilibren latencia y energía. Un sistema puede usar interrupciones durante períodos de baja actividad y activar polling cuando detecta carga sostenida. Esta adaptación permite evitar el desperdicio energético del sondeo permanente sin incurrir siempre en la latencia de interrupciones.

Antes de continuar con el análisis, la siguiente tabla sintetiza los principales criterios que permiten comparar interrupciones, polling y estrategias híbridas en arquitecturas de E/S actuales. La tabla no pretende establecer una superioridad universal, sino mostrar que cada mecanismo responde a un contexto operativo distinto, condicionado por latencia, throughput, consumo, complejidad y tipo de carga.

Tabla 8

Comparación arquitectónica entre interrupciones, polling y estrategias híbridas de E/S

Criterio de análisis	Interrupciones de hardware	Sondeo o polling	Estrategias híbridas
Principio de funcionamiento	El dispositivo notifica a la CPU cuando requiere atención	La CPU revisa activamente colas, registros o descriptores	El sistema alterna entre interrupciones, polling, coalescencia o busy polling según carga
Ventaja principal	Ahorra ciclos de CPU cuando los eventos son esporádicos	Reduce latencia de detección y evita sobrecosto de interrupciones frecuentes	Equilibra latencia, throughput y consumo energético
Limitación principal	Alto costo cuando la tasa de eventos es muy elevada	Consume CPU incluso sin trabajo disponible	Requiere umbrales, políticas adaptativas y monitoreo de carga
Impacto en latencia	Puede introducir variabilidad por atención del kernel y planificación	Puede ofrecer latencia menor y más predecible	Puede reducir latencia bajo carga y ahorrar energía en reposo
Impacto en consumo energético	Favorece ahorro cuando hay baja actividad	Mantiene núcleos activos y puede elevar consumo	Reduce consumo respecto al polling permanente
Relación con NVMe y SPDK	Menos adecuada para cargas NVMe	Muy usada en SPDK y user-level I/O para reducir sobrecarga	Útil cuando la carga varía entre períodos de reposo y alta intensidad

	extremadamente intensivas		
Relación con redes de alta velocidad	Puede saturar la CPU si se interrumpe por cada paquete	Favorece procesamiento sostenido de paquetes en colas de alta tasa	Coalescencia y polling adaptativo reducen tormentas de interrupciones
Relación con virtualización	Puede requerir mediación, inyección o traducción por hipervisor	Puede reducir parte de la mediación si se combina con colas compartidas o passthrough	Permite ajustar desempeño según VM, contenedor o servicio
Riesgo principal	Tormentas de interrupciones, cambios de contexto y perturbación de caché	Desperdicio de ciclos, menor eficiencia energética y ocupación de núcleos	Complejidad de configuración y posible comportamiento inestable si los umbrales son inadecuados
Escenarios recomendados	Dispositivos de baja frecuencia, eventos esporádicos, sistemas energéticamente sensibles	NVMe intensivo, packet processing, HPC, baja latencia y colas siempre activas	Cloud, edge, almacenamiento remoto, redes variables y cargas mixtas

Nota**.** Elaboración propia a partir de Zhu et al. (2023), Farshin et al. (2020), Kashyap y Lu (2022), Sharan et al. (2023) y Smolyar et al. (2020).

La tabla permite observar que el criterio central no es elegir entre interrupciones y polling de manera absoluta, sino definir el mecanismo adecuado según la intensidad y sensibilidad de la carga. Las interrupciones conservan valor cuando los eventos son infrecuentes o cuando la eficiencia energética es prioritaria. El polling es especialmente pertinente cuando las colas se mantienen ocupadas, la latencia debe minimizarse y existen núcleos disponibles para dedicarse a E/S. Las estrategias híbridas se vuelven indispensables cuando el sistema enfrenta variabilidad, porque permiten cambiar de comportamiento según la presión de trabajo. En arquitecturas cloud y edge, donde la carga puede variar por usuario, hora, ubicación o tipo de servicio, esta adaptabilidad resulta fundamental.

En sistemas NVMe, la elección entre interrupciones y polling se articula con el diseño de colas. NVMe permite múltiples colas de envío y completación, lo que favorece paralelismo y afinidad con núcleos. Cuando cada núcleo puede sondear su propia cola, se reducen bloqueos y se mejora escalabilidad. No obstante, esto exige asignar memoria, colas y núcleos de manera coherente. Si varias colas compiten por el mismo núcleo, si los buffers se ubican en memoria remota o si las solicitudes son demasiado pequeñas, el beneficio puede disminuir. La arquitectura de E/S debe, por tanto, alinear colas, memoria, CPU y dispositivo.

En redes de alta velocidad, la atención de eventos se vincula con la administración de lotes. Procesar un paquete por evento puede resultar ineficiente; procesar lotes reduce overhead, pero aumenta latencia individual. Esta tensión se observa tanto en interrupciones coalescentes como en polling. Un bucle de sondeo puede procesar varios descriptores por iteración para amortizar costo, pero si espera demasiado para formar lotes, afecta tiempo de respuesta. En aplicaciones de red, esta decisión depende de si se prioriza throughput agregado, latencia mediana o latencia de cola. Farshin et al. (2020) advierten que, a velocidades multi-hundred-gigabit, la latencia de cola y la gestión de caché se vuelven críticas, por lo que la optimización de E/S debe atender no solo el promedio, sino también los peores casos.

La relación entre polling y DDIO es particularmente delicada. Si el núcleo que sondea procesa datos que la NIC depositó en la LLC, el rendimiento puede mejorar al evitar accesos a DRAM. Sin embargo, si los datos de E/S desplazan información útil o si el núcleo está en otro socket, el beneficio puede convertirse en penalización. Sharan, Vutukuru y Panda (2023) proponen simular DDIO junto con CPU, caché, memoria y procesamiento de red para estudiar estas interacciones de manera integral, debido a que las decisiones sobre caché, prefetching y jerarquía de memoria afectan directamente el rendimiento del packet processing. La conclusión es clara: el mecanismo de notificación no puede separarse de la ubicación física de los datos ni de la política de caché.

El polling en sistemas desagregados también debe considerar la distancia lógica entre aplicación y servicio. En NVMe-oF, una aplicación puede enviar solicitudes hacia un servicio de almacenamiento remoto. Si ambos componentes están co-localizados en el mismo host físico, usar una ruta de red completa es ineficiente. Si se encuentran en hosts distintos, la red es inevitable. Kashyap y Lu (2022) incorporan esta conciencia de localidad mediante canales de memoria compartida para tráfico intra-nodo y TCP optimizado para tráfico inter-nodo. Esta lógica es relevante más allá de NVMe: los sistemas modernos requieren mecanismos de E/S capaces de reconocer cuándo el dato está local, remoto, virtualizado, compartido o distribuido.

En edge computing, el polling debe evaluarse con cautela. Un nodo edge puede tener pocos núcleos, bajo consumo, restricciones térmicas y necesidad de operar con batería o hardware económico. Dedicar un núcleo a sondeo permanente puede no ser viable. No obstante, ciertas aplicaciones industriales o energéticas pueden requerir baja latencia y continuidad local, lo que justifica sondeo selectivo para eventos críticos. Ferreira et al. (2022) muestran que el middleware edge para gestión energética del hogar debe operar en hardware de bajo costo, con baja latencia, bajo error y consumo reducido de recursos, lo cual obliga a diseñar mecanismos de comunicación y procesamiento ajustados a capacidades limitadas. En este tipo de entorno, las estrategias híbridas suelen ser más apropiadas que el polling permanente.

La seguridad también interviene en esta discusión. El polling en espacio de usuario puede reducir latencia, pero al acercar la aplicación al dispositivo se deben preservar aislamiento y control. El sistema debe impedir que una aplicación lea o escriba buffers no autorizados, monopolice colas compartidas o interfiera con otras cargas. En entornos multi-tenant, la eficiencia de E/S no puede sacrificar aislamiento. Por ello, IOMMU, permisos de memoria, colas separadas, canales por aplicación y validación de descriptores son elementos indispensables. La virtualización y los contenedores deben ofrecer rutas rápidas sin abrir accesos indebidos al hardware.

En consecuencia, interrupciones y polling no representan solo dos técnicas de notificación, sino dos formas de organizar la relación temporal entre CPU y dispositivo. Las interrupciones responden al evento cuando aparece; el polling anticipa que el evento aparecerá y mantiene al procesador listo. La primera estrategia conserva recursos cuando el sistema está inactivo; la segunda reduce espera cuando el sistema está saturado. Las arquitecturas modernas, caracterizadas por cargas variables, dispositivos rápidos y entornos virtualizados, tienden a combinar ambas. La inteligencia del sistema reside en decidir cuándo esperar y cuándo buscar activamente.

En síntesis, la comparación entre interrupciones de hardware y sondeo demuestra que la entrada/salida moderna exige una administración contextual de la latencia. No existe un mecanismo universalmente óptimo. Las

interrupciones siguen siendo adecuadas para cargas ligeras y sistemas sensibles a energía; el polling resulta eficaz en E/S intensiva, NVMe, redes de alta velocidad y HPC; las estrategias híbridas son necesarias en cloud, edge y entornos de carga variable. El diseño arquitectónico debe considerar no solo el dispositivo, sino también la CPU, la caché, la memoria, la virtualización, la topología NUMA, el consumo energético y la criticidad temporal de la aplicación. De esta manera, la E/S deja de ser una simple comunicación periférica y se convierte en un sistema de decisión sobre cuándo, dónde y cómo atender el movimiento de datos.

El impacto de la virtualización en la abstracción de recursos físicos

La virtualización de hardware constituye una de las transformaciones más significativas en la organización computacional contemporánea, porque modifica la relación directa entre software y recursos físicos. En una máquina no virtualizada, el sistema operativo administra CPU, memoria, almacenamiento, red, interrupciones y dispositivos de entrada/salida como si fueran recursos propios del entorno físico donde se ejecuta. En cambio, en un sistema virtualizado, esos recursos son mediados por una capa adicional, el hipervisor o Virtual Machine Monitor, que abstrae, particiona y controla el acceso de múltiples máquinas virtuales al mismo hardware. Esta mediación permite consolidar cargas, aislar dominios de ejecución, mejorar utilización de servidores, facilitar despliegues cloud y soportar arquitecturas embebidas donde varios sistemas operativos deben coexistir sobre una misma plataforma. Sin embargo, esta abstracción no es gratuita: introduce costos de traducción, control, seguridad, administración de memoria y gestión de E/S que deben resolverse mediante cooperación entre hardware, firmware, hipervisor y sistema operativo invitado.

El hipervisor cumple una función arquitectónica de intermediación. Su tarea no se reduce a “crear máquinas virtuales”, sino a sostener la ilusión controlada de que cada sistema invitado posee recursos propios, aunque en realidad comparte CPU, memoria, buses, dispositivos y almacenamiento con otros entornos. Aalam, Kumar y Gour (2021) explican que la virtualización permite ejecutar múltiples máquinas sobre un mismo hardware físico mediante una capa de hipervisor que

administra los recursos y aísla los entornos invitados, pero también advierten que esa posición privilegiada lo convierte en un objetivo crítico para la seguridad del sistema. Esta doble condición define la naturaleza del hipervisor: es al mismo tiempo un mecanismo de eficiencia y una frontera de confianza. Si funciona correctamente, habilita consolidación y aislamiento; si falla o es comprometido, puede exponer todas las máquinas virtuales que dependen de él.

La virtualización puede organizarse, de forma general, en hipervisores tipo 1 y tipo 2. Los hipervisores tipo 1 se ejecutan directamente sobre el hardware físico, sin depender de un sistema operativo anfitrión. Por ello, suelen emplearse en centros de datos, cloud computing, servidores empresariales y plataformas donde se requieren alto rendimiento, aislamiento fuerte y control directo de recursos. Los hipervisores tipo 2 se ejecutan sobre un sistema operativo anfitrión, por lo que resultan más convenientes para entornos de escritorio, pruebas, desarrollo y uso educativo, aunque incorporan una capa adicional de mediación. Aalam et al. (2021) describen esta diferencia al señalar que los hipervisores nativos acceden directamente al hardware, mientras que los hospedados coordinan las solicitudes de CPU, memoria, disco y red a través del sistema operativo base. La distinción no es solo taxonómica; determina el nivel de control, el perfil de rendimiento, el modelo de seguridad y la complejidad de administración del sistema.

La virtualización de CPU exige que el hipervisor controle instrucciones privilegiadas, cambios de modo, temporizadores, interrupciones y asignación de tiempo de procesador. En arquitecturas clásicas, ciertas instrucciones ejecutadas por un sistema operativo invitado podían comportarse de forma problemática si no eran atrapadas correctamente por el hipervisor. Por ello, las arquitecturas modernas han incorporado soporte explícito de virtualización mediante extensiones de hardware. Estas extensiones permiten reducir el costo de trap-and-emulation, mejorar el aislamiento y facilitar que el sistema invitado ejecute muchas operaciones con menor intervención del hipervisor. En este sentido, la virtualización pasó de ser una técnica predominantemente software a convertirse en una propiedad de la arquitectura del procesador, expresada en modos de privilegio, registros de control, traducción de memoria y mecanismos de interrupción.

La virtualización de memoria es una de las dimensiones más complejas, porque cada máquina virtual opera con la expectativa de poseer su propio espacio de direcciones. El sistema operativo invitado traduce direcciones virtuales de procesos a direcciones físicas invitadas; el hipervisor, a su vez, debe traducir esas direcciones físicas invitadas a direcciones físicas reales de la máquina anfitriona. Esta traducción en dos etapas permite aislamiento y multiplexación de memoria, pero puede aumentar la latencia si no se apoya en estructuras microarquitectónicas eficientes. Sá et al. (2023) explican que la extensión Hypervisor de RISC-V incorpora el modelo de dos etapas, donde VS-stage traduce direcciones virtuales invitadas a físicas invitadas y G-stage traduce direcciones físicas invitadas a físicas del host, lo que exige una Nested-MMU capaz de coordinar ambas fases. La consecuencia es clara: virtualizar memoria no consiste únicamente en "repartir RAM", sino en mantener una cadena de traducciones rápida, protegida y coherente.

Los TLB desempeñan un papel decisivo en esta cadena. Si cada acceso a memoria tuviera que recorrer tablas de páginas de dos niveles completos, el rendimiento sería inaceptable. Por ello, los procesadores utilizan Translation Lookaside Buffers para almacenar traducciones recientes. En sistemas virtualizados, la presión sobre los TLB aumenta porque las traducciones incluyen identificadores de proceso, máquina virtual y etapa de traducción. Sá et al. (2023) proponen mejoras microarquitectónicas en CVA6, como un G-Stage TLB y un L2 TLB, para reducir el sobrecosto de virtualización en RISC-V; sus resultados muestran que una configuración equilibrada puede mejorar el desempeño de una máquina virtual Linux sobre Bao con un costo reducido en área y potencia. Este aporte evidencia que la eficiencia de la virtualización depende de decisiones finas de microarquitectura, no solo de la presencia nominal de una extensión ISA.

La virtualización de E/S introduce otro conjunto de problemas. Mientras que virtualizar CPU y memoria puede resolverse mediante modos de privilegio y traducciones, los dispositivos físicos poseen estados, colas, interrupciones, buffers, registros MMIO y capacidades DMA que no siempre son fáciles de compartir. Una máquina virtual puede necesitar acceder a una tarjeta de red, un disco NVMe o un acelerador, pero el hipervisor debe garantizar que ese acceso no interfiera con otras máquinas ni comprometa memoria ajena. Existen varias

estrategias: emulación de dispositivo, paravirtualización, passthrough, SR-IOV, mediación por software, colas compartidas y asignación directa protegida por IOMMU. Cada estrategia equilibra rendimiento, flexibilidad, aislamiento y capacidad de migración. La emulación favorece compatibilidad, pero suele ser lenta; el passthrough reduce sobrecosto, pero dificulta movilidad y exige protección estricta; la paravirtualización mejora eficiencia, pero requiere controladores especializados.

La figura siguiente organiza la virtualización como una pirámide de abstracción. Su propósito es mostrar que la aplicación se encuentra en la capa superior, alejada del hardware físico, mientras que cada nivel inferior proporciona soporte, mediación y control.

Figura 4. ***Pirámide de abstracción en virtualización de hardware y gestión de E/S***

Nota. *Elaboración propia a partir de Aalam et al. (2021), Sá et al. (2023), Nordholz (2020), Zhu et al. (2023) y Smolyar et al. (2020).*

La figura permite comprender que la virtualización no elimina el hardware, sino que reorganiza la forma en que el software lo percibe y lo utiliza. En la base se encuentran los recursos físicos: procesador, memoria, buses, DMA, almacenamiento, red y dispositivos. Sobre ellos opera la virtualización de CPU, memoria, interrupciones y E/S, que traduce recursos materiales en recursos administrables. El hipervisor se ubica como capa de control porque decide cómo se asignan, aíslan y supervisan esos recursos. La máquina virtual aparece como una construcción lógica que recibe del hipervisor un entorno aparentemente

propio. Sobre ella se ejecuta un sistema operativo invitado, y finalmente las aplicaciones operan sin conocer directamente la complejidad de las capas inferiores. Esta pirámide muestra la principal ventaja y el principal riesgo de la virtualización: cuanto más abstracto es el entorno, más flexible resulta para el software, pero mayor es la responsabilidad de las capas inferiores para preservar rendimiento, seguridad e integridad.

En sistemas intensivos en E/S, la virtualización puede amplificar costos de interrupciones, polling y DMA. Una interrupción física puede tener que ser capturada por el hipervisor y luego inyectada como interrupción virtual. Una operación DMA debe validarse para evitar que el dispositivo escriba en memoria perteneciente a otra máquina virtual. Una cola de almacenamiento puede atravesar capas de emulación o traducción antes de llegar al dispositivo real. Por ello, la virtualización moderna necesita soporte de hardware como IOMMU, VT-d, SR-IOV, extensiones de interrupción virtual, colas multi-queue y mecanismos de aislamiento de DMA. Sin estos mecanismos, la abstracción puede afectar gravemente la latencia y el throughput de aplicaciones de red, almacenamiento o HPC.

La seguridad de la virtualización depende de que el hipervisor mantenga fronteras firmes entre máquinas virtuales. Cada VM debe tener acceso únicamente a su memoria, sus dispositivos asignados, sus interrupciones virtuales y sus recursos autorizados. La dificultad es que el hipervisor opera con privilegios elevados y, por tanto, se convierte en parte crítica de la Trusted Computing Base. Aalam et al. (2021) advierten que, cuando un hipervisor se compromete, el atacante puede obtener acceso al conjunto de máquinas hospedadas en el mismo servidor, debido a que el hipervisor administra registros, memoria e I/O. Esta afirmación obliga a considerar que la virtualización no es seguridad automática. El aislamiento es una propiedad que debe diseñarse, verificarse y mantenerse frente a fallos, configuraciones incorrectas o ataques.

Nordholz (2020) aborda esta problemática desde una perspectiva distinta al proponer Phidias, un hipervisor embebido diseñado para reducir la dinámica en tiempo de ejecución y trasladar gran parte de la configuración al tiempo de compilación. Su argumento parte de que muchos sistemas embebidos no

necesitan la flexibilidad extrema de la nube, porque el número de tareas, recursos y máquinas aisladas puede conocerse desde el diseño. Al eliminar subsistemas dinámicos innecesarios, se reduce el TCB y se vuelve más tratable la verificación simbólica del binario final. Esta propuesta permite comprender que la virtualización no debe adoptar una forma única. En cloud se privilegia elasticidad; en sistemas embebidos críticos puede ser más importante la verificabilidad, la mínima dinámica y la reducción del espacio de estados.

El enfoque de Phidias es relevante porque cuestiona la idea de que toda virtualización debe maximizar flexibilidad en tiempo de ejecución. En un centro de datos, puede ser indispensable crear, migrar y destruir máquinas virtuales dinámicamente. En un sistema automotriz, industrial o médico, esa dinámica puede ser innecesaria e incluso perjudicial para la certificación. La arquitectura embebida puede beneficiarse de hipervisores estáticos que asignan memoria, dispositivos e interrupciones desde compilación, evitando asignadores dinámicos, estructuras cambiantes y estados difíciles de verificar. Así, la abstracción de recursos físicos puede orientarse a objetivos diferentes según el dominio: eficiencia económica en cloud, consolidación segura en servidores, aislamiento determinista en embebidos y baja latencia en HPC.

La virtualización también se relaciona con el principio de minimalidad. Un hipervisor pequeño reduce superficie de ataque, facilita auditoría y simplifica razonamiento formal. Sin embargo, un hipervisor demasiado mínimo puede carecer de funcionalidades necesarias para administración, migración, virtualización avanzada de dispositivos o soporte de múltiples plataformas. El diseño debe equilibrar minimalidad y utilidad. En sistemas de propósito general, la complejidad puede ser inevitable para soportar diversidad de hardware y cargas. En sistemas específicos, la reducción del runtime puede producir beneficios significativos. Nordholz (2020) muestra que, cuando la configuración del sistema se conoce de antemano, eliminar dinámica no esencial permite razonar directamente sobre el estado concreto del sistema, algo mucho más difícil en hipervisores generales con asignación y reconfiguración dinámica.

La virtualización en RISC-V incorpora una dimensión adicional de interés académico e industrial. RISC-V, por su apertura y modularidad, permite estudiar

de manera transparente cómo una ISA integra soporte de hipervisor y cómo las implementaciones pueden explorar diferentes configuraciones microarquitectónicas. Sá et al. (2023) destacan que la extensión Hypervisor de RISC-V define nuevos modos de privilegio, traducción en dos etapas y estructuras de control que permiten ejecutar sistemas invitados de forma más eficiente, pero también señalan que el rendimiento depende de optimizaciones como GTLB y L2 TLB. Esta relación entre especificación abierta e implementación concreta resulta valiosa para investigación porque permite examinar no solo qué exige la arquitectura, sino cómo distintas decisiones de hardware afectan potencia, área y desempeño.

La exploración PPA, es decir, rendimiento, potencia y área, es fundamental para evaluar virtualización asistida por hardware. Una mejora microarquitectónica puede reducir ciclos de ejecución de una VM, pero aumentar área o consumo. Un TLB adicional puede mejorar traducciones, pero ocupar recursos. Un mecanismo de virtualización de interrupciones puede reducir latencia, pero agregar lógica. En sistemas embebidos, estos costos pueden ser críticos; en servidores, pueden justificarse por consolidación; en edge, deben balancearse con energía y costo. Sá et al. (2023) muestran que una configuración con Sstc y GTLB de ocho entradas ofrece una mejora de rendimiento con bajo costo relativo de área y potencia, lo que ilustra la necesidad de seleccionar puntos de diseño equilibrados.

En la práctica, virtualizar recursos físicos implica gestionar una tensión permanente entre transparencia y control. La VM debe sentirse suficientemente transparente para que el sistema operativo invitado funcione sin modificaciones importantes, pero el hipervisor debe conservar control suficiente para garantizar aislamiento, reparto justo y seguridad. Cuando se usa emulación completa, la transparencia es alta, pero el rendimiento puede ser bajo. Cuando se usa paravirtualización, el rendimiento mejora, pero se requiere cooperación del invitado. Cuando se usa passthrough, el rendimiento puede aproximarse al nativo, pero disminuye la capacidad del hipervisor para mediar y migrar. Así, cada técnica define un punto distinto en el espacio de diseño.

Desde la perspectiva de la E/S, la virtualización no puede evaluarse sin considerar DMA. Si una máquina virtual controla un dispositivo capaz de escribir

directamente en memoria, debe existir una IOMMU que traduzca y restrinja las direcciones DMA. De lo contrario, un dispositivo asignado a una VM podría escribir fuera de su región autorizada. El DMA, que en sistemas no virtualizados se utiliza para acelerar transferencias, en sistemas virtualizados debe convertirse también en objeto de protección. Esto es especialmente relevante en NVMe, NICs, GPU y aceleradores, donde el acceso directo a memoria es central para el rendimiento. La arquitectura moderna debe permitir DMA rápido, pero no DMA irrestricto.

En este sentido, la virtualización redefine la noción de "recurso físico". Una CPU física se convierte en vCPU; la memoria física se convierte en memoria invitada respaldada por páginas reales; una interrupción física se convierte en interrupción virtual; un dispositivo puede convertirse en dispositivo emulado, paravirtualizado, compartido o asignado; una cola de E/S puede convertirse en una estructura mediada por el hipervisor. Cada traducción conserva una parte del comportamiento original, pero agrega reglas de control. La eficiencia del sistema depende de que estas traducciones sean suficientemente rápidas y de que el software invitado no perciba diferencias que comprometan su funcionamiento.

En síntesis, el impacto de la virtualización en la abstracción de recursos físicos es profundo porque transforma la arquitectura computacional en una estructura de capas controladas. El hardware deja de ser visto directamente por el sistema operativo y pasa a ser administrado por un hipervisor que crea entornos aislados, traduce direcciones, media interrupciones, asigna dispositivos y regula DMA. Esta abstracción permite cloud computing, consolidación, seguridad, sistemas embebidos mixtos y despliegues flexibles, pero introduce desafíos de rendimiento, seguridad, verificabilidad y consumo. La virtualización moderna no puede entenderse como una simple técnica de simulación; es una forma de organización del hardware compartido, donde la eficiencia depende de la calidad de la mediación entre recursos físicos y entornos lógicos.

Arquitecturas orientadas a la nube

Las arquitecturas orientadas a la nube representan una reorganización profunda de la infraestructura computacional, porque desplazan el énfasis desde la

posesión local de recursos hacia la provisión flexible, escalable y administrada de capacidades de cómputo, almacenamiento, red, seguridad y software. En lugar de concebir cada sistema como una máquina aislada con recursos fijos, la nube propone un modelo en el que los recursos se agregan, virtualizan, automatizan y consumen bajo demanda. Esta transformación no solo modificó la forma de desplegar aplicaciones empresariales, científicas y educativas, sino también la manera de comprender la organización de computadores, porque el hardware físico se vuelve parte de una infraestructura abstracta, distribuida y gobernada por capas de virtualización, orquestación, monitoreo, balanceo y seguridad. Sehgal, Bhatt y Acken (2023) describen la computación en la nube como un cambio arquitectónico sustentado en virtualización, multi-tenancy, aprovisionamiento bajo demanda, orientación a servicios y acuerdos de nivel de servicio, lo que permite interpretar la nube como una evolución de la computación hacia un modelo de utilidad tecnológica.

La nube centralizada se consolidó porque permitió concentrar recursos en centros de datos altamente administrados, capaces de ofrecer elasticidad, redundancia, escalabilidad y eficiencia operativa. Bajo este modelo, las organizaciones no necesitan adquirir de forma permanente toda la infraestructura necesaria para sus picos de demanda, sino que pueden aprovisionar recursos dinámicamente. Esta elasticidad se volvió esencial para aplicaciones web, plataformas de datos, inteligencia artificial, almacenamiento empresarial, servicios educativos y sistemas de información de salud. No obstante, la nube centralizada también introduce dependencias críticas: conectividad permanente, latencia de red, concentración de datos sensibles, riesgo de punto único de falla lógico o regional, y costos acumulativos asociados al almacenamiento, transferencia y procesamiento. Por ello, la arquitectura cloud debe analizarse como una solución poderosa, pero no universalmente suficiente para todos los dominios.

En sectores como salud, la elección entre arquitecturas centralizadas y descentralizadas revela con claridad esta tensión. Abughazalah, Alsaggaf, Saifuddin y Sarhan (2024) explican que las arquitecturas cloud centralizadas favorecen integración, acceso rápido, simplicidad operativa y consolidación de información, pero enfrentan desafíos de privacidad, confidencialidad y punto

único de falla; en cambio, las arquitecturas descentralizadas distribuyen datos entre varios nodos, mejoran resiliencia y reducen latencia, aunque incrementan complejidad de integración, gestión y costo operativo. Aunque su análisis se sitúa en sistemas de intercambio de información sanitaria, el principio es aplicable a muchos sistemas contemporáneos: concentrar recursos facilita administración y control, mientras distribuirlos mejora proximidad y resiliencia, pero exige coordinación más compleja.

La virtualización fue la base técnica que permitió a la nube consolidar múltiples cargas sobre la misma infraestructura física. Al convertir servidores, almacenamiento y redes en recursos lógicos, los proveedores pudieron asignar capacidad a diferentes usuarios sin que estos conocieran la organización física subyacente. Esta abstracción habilitó multi-tenancy, migración de máquinas virtuales, recuperación ante fallos, escalamiento horizontal y administración centralizada. Sin embargo, el modelo basado exclusivamente en máquinas virtuales también presenta costos de arranque, sobrecarga de sistema operativo invitado y mayor consumo de recursos frente a alternativas más ligeras. Por ello, la evolución cloud incorporó contenedores, microservicios, funciones serverless y arquitecturas cloud-native, que buscan aumentar modularidad, velocidad de despliegue y eficiencia operativa.

Los contenedores introdujeron una forma más ligera de empaquetar aplicaciones y sus dependencias, compartiendo el kernel del sistema anfitrión y reduciendo el peso de la virtualización completa. Esta tecnología facilitó la expansión de microservicios, donde una aplicación deja de ser un bloque monolítico y se organiza en servicios pequeños, independientes y comunicados mediante APIs. Esta descomposición permite actualizar, escalar y desplegar componentes de manera diferenciada, pero también introduce desafíos de comunicación, observabilidad, consistencia, seguridad y orquestación. La arquitectura cloud-native surge precisamente para gestionar esta complejidad mediante contenedores, orquestadores, infraestructura declarativa, automatización, balanceo, monitoreo, service mesh y prácticas de integración continua.

Los microservicios son especialmente relevantes porque modifican la relación entre aplicación y arquitectura. En un diseño monolítico, el sistema suele

desplegarse como una unidad completa; en un diseño de microservicios, cada función puede tener su propio ciclo de vida, su propia escala, su propio almacenamiento y su propio patrón de comunicación. Esta fragmentación permite agilidad, pero convierte la red en parte interna de la aplicación. Las llamadas entre componentes, antes locales dentro de un proceso, pueden transformarse en comunicaciones entre servicios, contenedores, nodos o regiones. Por ello, el rendimiento de una aplicación cloud-native depende no solo del código, sino de latencia de red, serialización, balanceo, descubrimiento de servicios, políticas de seguridad y observabilidad distribuida.

La computación serverless profundiza esta tendencia al ocultar aún más la administración de servidores. En este modelo, el desarrollador despliega funciones o unidades lógicas de ejecución, mientras la plataforma administra aprovisionamiento, escalamiento, aislamiento y cobro por uso. Esta arquitectura puede ser muy eficiente para cargas event-driven, procesamiento por demanda, automatización y tareas intermitentes, porque evita mantener servidores activos cuando no hay trabajo. No obstante, también presenta limitaciones: latencia de arranque en frío, dependencia fuerte del proveedor, restricciones de tiempo de ejecución, complejidad para depurar flujos distribuidos y dificultades para cargas persistentes o de baja latencia. La organización computacional serverless muestra hasta qué punto la nube ha convertido el hardware en una infraestructura invisible, aunque no inexistente.

Las arquitecturas distribuidas contemporáneas articulan nube, paralelismo, edge, blockchain, inteligencia artificial distribuida y sistemas heterogéneos. Dai, Hossain y Wang (2025) sostienen que los sistemas paralelos y distribuidos han evolucionado hacia modelos capaces de responder a demandas de escalabilidad, eficiencia y procesamiento masivo, incorporando tendencias como computación heterogénea, neuromórfica, cloud-native, serverless, blockchain y aprendizaje distribuido. Esta visión permite situar la nube como parte de un ecosistema más amplio donde los recursos ya no se organizan únicamente en centros de datos centrales, sino también en nodos edge, dispositivos inteligentes, aceleradores especializados y plataformas distribuidas geográficamente.

El edge computing aparece como respuesta a los límites del cloud centralizado cuando las aplicaciones requieren baja latencia, continuidad operativa, reducción de tráfico o preservación local de datos sensibles. En lugar de enviar toda la información hacia centros de datos remotos, el edge ubica procesamiento cerca de los dispositivos que producen o consumen datos. Harjula, Artemenko y Forsström (2022) explican que el edge computing permite desplegar tareas computacionales cerca de los nodos finales, reduciendo latencia y vulnerabilidad frente a problemas de red, lo cual resulta crítico en Industrial IoT, 5G, automatización, fábricas inteligentes, realidad aumentada y aplicaciones con requisitos estrictos de confiabilidad. Esta proximidad redefine el flujo de datos: no todo debe viajar a la nube, porque parte del valor puede extraerse localmente.

Fog computing y mist computing amplían esta discusión al proponer capas intermedias entre dispositivos finales y nube central. Mientras el edge se refiere a recursos computacionales próximos a usuarios o sensores, el fog suele describir una arquitectura lógica distribuida que permite procesamiento, almacenamiento y análisis en nodos intermedios. Mist computing, por su parte, se asocia al procesamiento en el extremo más cercano al dispositivo, incluso en nodos de capacidad muy limitada. Estas categorías muestran que el sistema computacional moderno se organiza como un continuo de proximidad: dispositivo, mist, edge, fog, cloud regional y cloud central. Cada capa procesa una parte distinta según latencia, capacidad, privacidad y costo.

En sistemas industriales, esta distribución es más que una mejora de rendimiento. La fábrica inteligente, el mantenimiento predictivo, los vehículos guiados autónomos, la realidad aumentada industrial y el control de procesos requieren respuestas que no siempre toleran el viaje completo hacia la nube. La latencia no es solo una métrica técnica, sino una condición de seguridad, continuidad y calidad operativa. Harjula et al. (2022) destacan que muchos casos industriales demandan ultra alta disponibilidad, confiabilidad, predictibilidad, baja latencia y comportamiento temporal determinista, por lo que el edge debe combinarse con tecnologías como 5G, virtualización en tiempo real, redes definidas por software y comunicación sensible al tiempo. La arquitectura de nube, por tanto, no desaparece, sino que se complementa con infraestructura periférica especializada.

Los sistemas de gestión energética del hogar muestran una aplicación concreta de esta lógica distribuida. Ferreira et al. (2022) proponen un middleware para Home Energy Management Systems basado en microservicios y edge computing, diseñado para ejecutarse en hardware de bajo costo con limitaciones de procesamiento y almacenamiento. Esta propuesta revela que el edge no solo pertenece a fábricas o telecomunicaciones, sino también a hogares, edificios, redes eléctricas y escenarios donde la proximidad al usuario permite operar aun con conectividad limitada. Además, el uso de microservicios en hardware restringido muestra que los principios cloud-native pueden adaptarse a entornos periféricos, aunque exijan mayor cuidado sobre consumo de recursos.

El modelo edge-cloud también transforma la gestión de datos. En una arquitectura centralizada, los datos suelen recolectarse en origen y enviarse a una nube donde se almacenan y procesan. En una arquitectura distribuida, los datos pueden filtrarse, agregarse, anonimizarse, inferirse o descartarse localmente antes de llegar a la nube. Esto reduce ancho de banda, mejora privacidad y disminuye latencia, pero exige decidir qué datos deben permanecer en el borde, cuáles deben sincronizarse y cuáles requieren procesamiento central. La arquitectura deja de ser un canal unidireccional hacia la nube y se convierte en un sistema de decisiones sobre ubicación del procesamiento.

La inteligencia artificial intensifica esta necesidad de distribución. Muchos modelos requieren entrenamiento en centros de datos con gran capacidad computacional, pero su inferencia puede desplegarse en dispositivos edge para responder localmente. El caso de ISMSFuse ilustra esta transición en agricultura inteligente, ya que Zhang et al. (2024) integran información de imagen y espectro mediante un modelo ligero adaptable a Raspberry Pi, buscando reconocimiento de enfermedad bacteriana en arroz bajo condiciones de edge computing. Esta orientación resulta relevante para el capítulo porque muestra que el diseño de algoritmos y arquitecturas debe considerar simultáneamente precisión, costo computacional, sensores disponibles, latencia y viabilidad de despliegue local.

La nube también se está reconfigurando por la presión energética de la inteligencia artificial. Los centros de datos dedicados a entrenamiento e inferencia consumen cantidades crecientes de electricidad, lo que obliga a

explorar hardware más eficiente y arquitecturas especializadas. Vogginger et al. (2024) plantean que el hardware neuromórfico, inspirado en el funcionamiento del cerebro y basado en procesamiento event-driven, ofrece una vía prometedora para reducir la energía necesaria en cargas de IA, aunque su incorporación en centros de datos exige resolver problemas de integración con software, algoritmos, modelos y operación cloud. Esta línea de investigación sugiere que las arquitecturas orientadas a la nube del futuro no estarán formadas únicamente por CPU y GPU, sino por recursos heterogéneos que incluirán aceleradores especializados y posiblemente sistemas neuromórficos.

El hardware neuromórfico se aparta del modelo clásico de procesamiento continuo porque trabaja con eventos, picos o activaciones esparsas. Esta aproximación puede ser energéticamente eficiente para tareas de percepción, sensores, reconocimiento de patrones y procesamiento temporal. Oladele (2024) señala que las redes neuronales de picos y los sistemas neuromórficos buscan emular principios de comunicación neuronal mediante procesamiento discreto, paralelo y de bajo consumo, con aplicaciones potenciales en IoT, robótica, sistemas autónomos y procesamiento sensorial. Aunque la madurez industrial de estas tecnologías aún es desigual, su inclusión en la discusión cloud es necesaria porque la sostenibilidad de la IA no puede depender indefinidamente del escalamiento energético de arquitecturas convencionales.

En síntesis, las arquitecturas orientadas a la nube han evolucionado desde centros de datos centralizados hacia un continuo distribuido de recursos computacionales. La nube aporta elasticidad, abstracción, multi-tenancy, servicios administrados y escalabilidad. Los contenedores y microservicios aportan modularidad y despliegue rápido. Serverless profundiza la abstracción operativa. El edge, fog y mist reducen latencia y acercan procesamiento al dato. Los aceleradores y hardware neuromórfico responden a demandas de rendimiento y eficiencia energética. Esta transformación redefine la organización de computadores porque el sistema ya no se limita a una placa, un servidor o una memoria local; se extiende como una arquitectura de capas distribuidas donde el valor computacional depende de ubicar cada carga en el punto más adecuado del continuo hardware-software-red.

Discusión crítica

El desplazamiento del procesamiento hacia la periferia mediante edge computing obliga a revisar críticamente la arquitectura de entrada/salida, porque el modelo tradicional, centrado en enviar datos hacia una nube remota para su almacenamiento y procesamiento, resulta insuficiente frente a aplicaciones que requieren baja latencia, continuidad operativa, privacidad contextual y respuesta inmediata. Durante años, la nube centralizada ofreció una solución eficiente para concentrar cómputo, almacenamiento y servicios bajo demanda; sin embargo, el crecimiento de IoT, IIoT, inteligencia artificial distribuida, sensores inteligentes, vehículos autónomos, agricultura de precisión, salud digital y gestión energética ha evidenciado que no todos los datos deben recorrer el mismo trayecto arquitectónico. En muchas aplicaciones contemporáneas, el valor del dato depende de su procesamiento oportuno cerca del lugar donde se genera. Por ello, la E/S deja de ser solo un mecanismo para transferir información entre periféricos y memoria, y se convierte en un problema estratégico de ubicación, filtrado, priorización, seguridad y decisión.

La tensión entre cloud centralizado y edge distribuido no debe interpretarse como una oposición absoluta, sino como una reconfiguración del continuo computacional. La nube conserva ventajas notables para entrenamiento de modelos, almacenamiento histórico, integración institucional, escalabilidad y administración centralizada; el edge, en cambio, aporta proximidad, menor latencia, resiliencia ante desconexiones y reducción de tráfico hacia centros de datos. Harjula, Artemenko y Forsström (2022) sostienen que las aplicaciones industriales requieren disponibilidad, confiabilidad, baja latencia y comportamiento predecible, condiciones que no siempre pueden garantizarse si toda la lógica se ubica en centros de datos distantes. Esta observación permite afirmar que el problema no consiste en reemplazar la nube por la periferia, sino en diseñar arquitecturas capaces de decidir qué debe ejecutarse localmente, qué debe permanecer distribuido y qué debe consolidarse en la nube.

La primera limitación crítica del edge es que acerca el procesamiento al dato, pero lo hace en entornos donde los recursos suelen ser más restringidos. Un servidor cloud puede contar con abundante memoria, aceleradores, refrigeración,

redundancia y administración profesional; un nodo edge puede operar sobre hardware de bajo costo, energía limitada, conectividad variable y menor capacidad de almacenamiento. Ferreira et al. (2022) muestran que los sistemas de gestión energética del hogar requieren middleware capaz de operar en hardware limitado, con baja latencia y bajo consumo de recursos, lo cual confirma que el edge exige diseños más cuidadosos que la simple migración de servicios cloud hacia dispositivos pequeños. La periferia no es una nube reducida; es un espacio arquitectónico con restricciones propias, donde cada decisión de E/S, memoria, comunicación y procesamiento tiene impacto directo sobre la viabilidad del sistema.

La segunda tensión se relaciona con el movimiento de datos. Muchas arquitecturas siguen diseñándose bajo la premisa de que el dato puede moverse con facilidad desde sensores hacia memoria, desde memoria hacia CPU, desde CPU hacia GPU, desde GPU hacia almacenamiento o desde nodos periféricos hacia la nube. Sin embargo, el movimiento de datos se ha convertido en uno de los principales costos de rendimiento y energía. Farshin, Roozbeh, Maguire y Kostić (2020) muestran que en redes de 100 y 200 Gbps la gestión de caché para datos de E/S condiciona la latencia y el throughput, debido a que acceder a memoria principal puede ser demasiado costoso frente al ritmo de llegada de paquetes. Esta problemática se amplifica en el edge, donde el ancho de banda, la energía y la memoria suelen ser más limitados. En consecuencia, las nuevas arquitecturas de E/S deben reducir copias, evitar trayectorias redundantes y procesar datos cerca de su fuente siempre que sea posible.

El DMA aparece aquí como una tecnología ambivalente. Por un lado, permite liberar a la CPU de transferencias repetitivas y habilita rutas de datos de alto rendimiento entre dispositivos, memoria, almacenamiento y aceleradores. Por otro, introduce riesgos de seguridad, coherencia y aislamiento, especialmente en sistemas virtualizados y multi-tenant. Un dispositivo con capacidad DMA puede escribir directamente en memoria, lo que exige mecanismos de protección como IOMMU, validación de buffers, asignación segura de páginas y control de permisos. En un entorno edge-cloud, donde dispositivos, contenedores, máquinas virtuales y servicios pueden compartir infraestructura, el DMA eficiente pero no controlado puede convertirse en una vulnerabilidad. La

arquitectura debe resolver una contradicción delicada: permitir rutas directas para reducir latencia sin debilitar las fronteras de aislamiento que hacen posible la virtualización y la multi-tenencia.

La virtualización también exhibe una tensión estructural. En cloud computing, la virtualización ha permitido consolidar cargas, abstraer recursos, mejorar utilización y ofrecer servicios bajo demanda. No obstante, cada capa de abstracción introduce posibles costos en traducción de memoria, interrupciones, E/S y seguridad. Sá et al. (2023) muestran que la virtualización en RISC-V exige soporte microarquitectónico específico, como GTLB y L2 TLB, para reducir el sobrecosto de la traducción en dos etapas y mejorar la ejecución de máquinas virtuales. Esto demuestra que la virtualización no puede depender solo del software. Para ser viable en sistemas de alta demanda o recursos limitados, requiere asistencia del hardware. En edge computing, esta condición es aún más crítica porque los márgenes de potencia, área y latencia son menores que en grandes centros de datos.

La seguridad de los hipervisores plantea otro problema crítico. Al convertirse en capa de mediación privilegiada, el hipervisor concentra control sobre CPU, memoria, E/S y dispositivos. Aalam, Kumar y Gour (2021) advierten que la vulnerabilidad del hipervisor amplía la superficie de ataque, porque una falla en esta capa puede comprometer las máquinas virtuales alojadas en el mismo hardware. Esta condición resulta especialmente sensible en arquitecturas distribuidas y periféricas, donde pueden coexistir servicios de diferentes niveles de criticidad sobre nodos físicamente accesibles o menos protegidos que un centro de datos. La promesa del edge como espacio de procesamiento local solo será sostenible si sus mecanismos de virtualización, aislamiento y actualización son suficientemente robustos.

La propuesta de hipervisores estáticos o con mínima dinámica, como Phidias, introduce una perspectiva crítica frente al modelo cloud dominante. Nordholz (2020) plantea que, en sistemas embebidos donde la configuración de tareas y recursos se conoce desde el diseño, eliminar dinámica innecesaria reduce el TCB y hace más tratable la verificación simbólica del binario final. Esta idea es relevante porque cuestiona la tendencia a trasladar la flexibilidad cloud a todos

los dominios. No toda arquitectura necesita crear y destruir recursos dinámicamente; algunas necesitan ser previsibles, verificables y certificables. En sistemas industriales, automotrices, médicos o energéticos, la capacidad de demostrar aislamiento y estabilidad puede ser más importante que la elasticidad extrema.

La comparación entre interrupciones y polling muestra otra tensión no resuelta. Las interrupciones ahorran CPU cuando los eventos son esporádicos, pero pueden saturar el sistema bajo cargas intensivas. El polling reduce latencia y evita sobrecostos de interrupción, pero consume ciclos y energía incluso cuando no hay trabajo disponible. En centros de datos de alto rendimiento, dedicar núcleos al sondeo puede ser aceptable; en nodos edge de bajo consumo, puede resultar inviable. Zhu, Wang, Xiao y Qin (2023) evidencian que los marcos de E/S de usuario, como SPDK, reducen cambios de contexto e interrupciones, pero también requieren una gestión eficiente de memoria fijada para no trasladar el cuello de botella hacia otro punto de la ruta de datos. La lección crítica es que la optimización de E/S suele desplazar los problemas, no eliminarlos por completo.

Las estrategias adaptativas ofrecen una vía intermedia, pero también introducen complejidad. NVMe-oAF, por ejemplo, propone seleccionar entre memoria compartida y transporte TCP optimizado según localidad y condiciones de comunicación. Kashyap y Lu (2022) muestran que esta adaptabilidad puede mejorar ancho de banda y latencia en nubes HPC con almacenamiento desagregado. Sin embargo, los sistemas adaptativos requieren monitoreo, umbrales, políticas de decisión y mecanismos de recuperación cuando las condiciones cambian. La adaptabilidad puede mejorar el rendimiento, pero también puede introducir comportamientos difíciles de predecir, especialmente si se combina con virtualización, contenedores, balanceo de carga y redes variables.

El acceso directo entre aceleradores y almacenamiento, como GPUDirect Storage, representa otra frontera crítica. Bayati, Leeser y Mi (2020) muestran que permitir a la GPU acceder directamente a NVMe mediante DMA peer-to-peer puede reducir pasos intermedios por memoria del host y mejorar el rendimiento en procesamiento Big Data acelerado. No obstante, esta optimización exige

repensar la función de la CPU, la coherencia de memoria, la seguridad de las rutas directas y la administración de errores. Cuanto más se habilitan caminos directos entre dispositivos, más se debilita la idea clásica de que la CPU y el sistema operativo son mediadores centrales de todo flujo de datos. Esto no significa que pierdan importancia, sino que su papel se desplaza hacia la coordinación, la autorización y la supervisión.

La arquitectura orientada a la nube también enfrenta una tensión de sostenibilidad. La expansión de inteligencia artificial, análisis masivo de datos y servicios permanentes incrementa el consumo energético de centros de datos. Vogginger et al. (2024) plantean que el hardware neuromórfico puede contribuir a una IA más eficiente energéticamente, pero también reconocen que su adopción en data centers requiere resolver desafíos de integración con software, modelos, algoritmos y operación cloud. Esta advertencia evita una lectura tecnológicamente ingenua: los aceleradores emergentes no transforman la arquitectura por sí solos. Solo producen impacto cuando se integran en una pila completa de programación, orquestación, datos, monitoreo y mantenimiento.

El edge computing tampoco debe idealizarse como solución automática de sostenibilidad. Procesar localmente puede reducir tráfico hacia la nube y mejorar latencia, pero también multiplica el número de dispositivos desplegados, aumenta la heterogeneidad, complica actualizaciones, distribuye riesgos de seguridad y puede generar infraestructuras difíciles de administrar. Un centro de datos centralizado puede optimizar refrigeración, energía y utilización de hardware; miles de nodos edge pueden operar con menor eficiencia individual y mayor complejidad de ciclo de vida. Por ello, la sostenibilidad debe evaluarse en todo el sistema: fabricación de dispositivos, consumo energético, transmisión de datos, vida útil del hardware, mantenimiento, reemplazo y capacidad de reutilización.

La privacidad es otro argumento frecuente a favor del edge, pero también requiere matices. Procesar datos localmente puede reducir exposición en tránsito y evitar enviar información sensible a la nube. En salud, energía, agricultura o industria, esta proximidad puede ser valiosa. Abughazalah et al. (2024) muestran que las arquitecturas descentralizadas pueden fortalecer privacidad y resiliencia

al distribuir datos entre nodos, aunque también incrementan complejidad de integración, gestión y costos. El dato local no está automáticamente protegido. Un nodo edge mal administrado, físicamente accesible o sin actualizaciones puede ser más vulnerable que un centro de datos con controles robustos. La privacidad depende de cifrado, control de acceso, políticas de retención, aislamiento y gobernanza, no solo de la ubicación física del procesamiento.

Desde una perspectiva arquitectónica, el principal desafío consiste en diseñar una E/S consciente del contexto. Los sistemas actuales deben decidir si conviene interrumpir o sondear, copiar o compartir, enviar a nube o procesar localmente, virtualizar o asignar directamente, replicar o reconstruir, acelerar con GPU o ejecutar en CPU, usar memoria compartida o red, mantener datos en caché o evitar contaminación. Cada decisión depende de latencia, ancho de banda, energía, seguridad, topología, carga y criticidad. No existe una solución universal. La organización computacional contemporánea se orienta hacia políticas dinámicas y arquitecturas híbridas capaces de adaptar sus rutas de datos al contexto operativo.

Esta complejidad transforma también la formación en arquitectura de computadores. Enseñar E/S como un conjunto de periféricos conectados por buses resulta insuficiente. Es necesario abordar DMA, IOMMU, caché de E/S, DDIO, NVMe, polling, interrupciones, virtualización, hipervisores, edge, cloud, contenedores y aceleradores como partes de una misma problemática: la gestión segura y eficiente del movimiento de datos. Los estudiantes deben comprender que el rendimiento no depende solo de la CPU ni de la memoria principal, sino de trayectorias completas entre sensores, dispositivos, redes, buffers, cachés, aceleradores, máquinas virtuales y servicios distribuidos.

En términos críticos, la próxima generación de arquitecturas de E/S deberá superar tres reduccionismos. El primero es creer que más ancho de banda resuelve todo; en realidad, sin localidad, seguridad y gestión de caché, más ancho de banda puede aumentar congestión y variabilidad. El segundo es asumir que la virtualización siempre introduce sobrecosto inevitable; con soporte adecuado de hardware, colas eficientes e IOMMU, puede acercarse al rendimiento nativo en ciertos contextos. El tercero es considerar que edge computing reemplaza a la

nube; más bien, la complementa dentro de un continuo donde cada capa cumple funciones distintas. La madurez arquitectónica consistirá en articular estas capas sin imponer a todas las aplicaciones el mismo patrón de procesamiento.

La discusión permite sostener que el desplazamiento hacia la periferia demanda nuevas arquitecturas de E/S porque el dato ya no sigue una ruta lineal. Puede generarse en un sensor, procesarse parcialmente en un nodo edge, almacenarse temporalmente en memoria local, inferirse mediante un acelerador, sincronizarse con la nube, replicarse por seguridad, reentrenar un modelo central y volver como actualización hacia miles de dispositivos. Cada etapa requiere decisiones sobre entrada/salida, DMA, interrupciones, polling, seguridad, virtualización y consistencia. La arquitectura del futuro será menos una máquina aislada y más una red coordinada de rutas de datos.

En síntesis, las tensiones entre cloud y edge, DMA y seguridad, virtualización y rendimiento, polling y energía, aceleración y movimiento de datos, sostenibilidad y escalabilidad muestran que la entrada/salida se ha convertido en el centro crítico de la organización computacional contemporánea. La eficiencia ya no se define únicamente por ejecutar más instrucciones por segundo, sino por reducir recorridos innecesarios, preservar aislamiento, ubicar procesamiento cerca del dato, adaptar la comunicación a la carga y sostener operación confiable en entornos distribuidos. Las nuevas arquitecturas de E/S deberán ser contextuales, heterogéneas, seguras y energéticamente conscientes. Solo así podrán responder a las exigencias de la próxima década, en la que la computación se desplegará simultáneamente en centros de datos, nodos edge, dispositivos inteligentes, aceleradores especializados y sistemas virtualizados de alta criticidad.

III. CONCLUSIONES

En conclusión, los paradigmas emergentes de entrada/salida, DMA y virtualización de hardware evidencian que la arquitectura computacional contemporánea ya no puede entenderse únicamente desde la potencia del procesador o la capacidad de la memoria principal. El rendimiento, la seguridad y la escalabilidad de los sistemas actuales dependen de la forma en que los datos se desplazan entre dispositivos, memorias, aceleradores, máquinas virtuales,

redes, nodos edge y plataformas cloud. Esta transformación obliga a situar la E/S como un componente central del diseño arquitectónico, porque el dato no solo debe ser procesado, sino también capturado, transferido, protegido, ubicado, virtualizado y entregado oportunamente.

El análisis de la gestión de periféricos y del acceso directo a memoria permitió reconocer que el DMA ha evolucionado desde una técnica orientada a liberar a la CPU de transferencias repetitivas hacia una arquitectura compleja del movimiento de datos. En los sistemas actuales, DMA se articula con NVMe, DDIO, NUDMA, user-level I/O, memoria fijada, listas scatter/gather, acceso directo entre GPU y almacenamiento, y rutas adaptativas para almacenamiento remoto. Esta evolución muestra que el desafío no consiste únicamente en mover datos más rápido, sino en moverlos por trayectorias más cortas, seguras y conscientes de la localidad. La eficiencia de un sistema intensivo en datos depende tanto de su capacidad de cómputo como de la inteligencia con que organiza sus rutas de transferencia.

La comparación entre interrupciones de hardware y sondeo permitió establecer que la atención de eventos de E/S exige una administración contextual de la latencia. Las interrupciones siguen siendo valiosas cuando los eventos son esporádicos, cuando el ahorro energético es prioritario o cuando el sistema no puede dedicar núcleos al monitoreo permanente. El polling, por su parte, resulta adecuado para cargas intensivas de almacenamiento, redes de alta velocidad y entornos HPC donde la latencia debe minimizarse y las colas permanecen activas. Las estrategias híbridas adquieren especial importancia en cloud y edge computing, porque permiten ajustar el comportamiento del sistema según carga, consumo, criticidad y disponibilidad de recursos.

El estudio de la virtualización permitió comprender que la abstracción de recursos físicos es una condición esencial de la computación moderna, pero también una fuente de tensiones arquitectónicas. El hipervisor permite consolidar cargas, aislar entornos, administrar memoria, virtualizar CPU, mediar interrupciones y regular el acceso a dispositivos, pero al mismo tiempo introduce costos de traducción, complejidad de seguridad y sobrecarga en la gestión de E/S. La virtualización asistida por hardware, las MMU anidadas, los TLB

especializados, las IOMMU y las técnicas de asignación directa de dispositivos muestran que la abstracción eficiente exige cooperación entre ISA, microarquitectura, sistema operativo e hipervisor.

Asimismo, la virtualización no debe interpretarse como un modelo único aplicable de forma idéntica a todos los contextos. En la nube, su valor principal se encuentra en la elasticidad, la consolidación, el multi-tenancy y la administración dinámica de recursos. En sistemas embebidos, industriales o críticos, puede ser más importante reducir la dinámica, minimizar la superficie de ataque, garantizar aislamiento temporal y espacial, y facilitar la verificación. Esta diferencia confirma que la arquitectura computacional debe diseñarse desde el contexto de uso, considerando si el sistema prioriza flexibilidad, rendimiento, seguridad, verificabilidad, energía o predictibilidad temporal.

El análisis de las arquitecturas orientadas a la nube mostró que la infraestructura computacional se ha desplazado hacia un continuo distribuido. La nube centralizada aporta elasticidad, escalabilidad, almacenamiento masivo, servicios administrados y capacidad de integración. Sin embargo, el edge computing, fog computing y mist computing responden a necesidades que la nube centralizada no siempre puede satisfacer: baja latencia, operación local, privacidad contextual, reducción del tráfico de red y continuidad ante fallos de conectividad. Por ello, el futuro no estará definido por una sustitución de la nube por el borde, sino por una articulación más inteligente entre centros de datos, nodos periféricos, dispositivos inteligentes y aceleradores especializados.

La discusión crítica permitió evidenciar que el desplazamiento del procesamiento hacia la periferia exige nuevas arquitecturas de E/S. Los datos ya no siguen una trayectoria simple desde un dispositivo hacia la memoria y de allí hacia la CPU. En los sistemas actuales, pueden generarse en sensores, filtrarse localmente, procesarse en un nodo edge, transferirse por DMA hacia un acelerador, sincronizarse con la nube, almacenarse en infraestructuras distribuidas y volver como resultado, actualización o decisión automatizada. Cada una de estas etapas demanda mecanismos de protección, control, baja latencia, selección de ruta y administración energética.

Finalmente, las tendencias de organización computacional para la próxima década apuntan hacia sistemas heterogéneos, distribuidos, virtualizados y energéticamente conscientes. Las arquitecturas eficientes deberán integrar DMA seguro, E/S consciente de localidad, virtualización asistida por hardware, polling adaptativo, almacenamiento desagregado, microservicios, contenedores, edge-cloud continuum, aceleradores especializados y, progresivamente, hardware neuromórfico u otras formas de cómputo no convencional. El objetivo ya no será únicamente incrementar la velocidad bruta, sino construir sistemas capaces de ubicar el procesamiento donde tenga mayor sentido, reducir el movimiento innecesario de datos, preservar la integridad de los recursos compartidos y sostener servicios confiables en infraestructuras cada vez más complejas. En ese horizonte, entrada/salida, DMA y virtualización dejan de ser temas periféricos de la arquitectura para convertirse en principios estructurales del diseño computacional moderno.

REFERENCIAS BIBLIOGRÁFICAS

Aalam, Z., Kumar, V., & Gour, S. (2021). A review paper on hypervisor and virtual machine security. Journal of Physics: Conference Series, 1950, Article 012027. https://doi.org/10.1088/1742-6596/1950/1/012027

Abughazalah, M., Alsaggaf, W., Saifuddin, S., & Sarhan, S. (2024). Centralized vs. decentralized cloud computing in healthcare. Applied Sciences, 14(17), Article 7765. https://doi.org/10.3390/app14177765

Agarwal, A. (1991). Limits on interconnection network performance. Massachusetts Institute of Technology, Laboratory for Computer Science.

Alam, M., & Varshney, A. K. (2015). A comparative study of interconnection network. International Journal of Computer Applications, 127(4), 37–43.

Alkhamisi, K. H. (2022). Cache coherence issues and solution: A review. International Journal of Information Systems and Computer Technologies, 1(2), 1–6.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. En AFIPS Conference Proceedings (Vol. 30, pp. 483–485). Association for Computing Machinery.

Bayati, M., Leeser, M., & Mi, N. (2020). Exploiting GPU direct access to non-volatile memory to accelerate Big Data processing [Conference paper].

Cao, Y., Wu, G., & Wang, H. (2011). A smart compression scheme for GPU-accelerated volume rendering of time-varying data. En 2011 International Conference on Virtual Reality and Visualization (pp. 205–210). IEEE. https://doi.org/10.1109/ICVRV.2011.56

Cho, Y.-S., Choi, E.-J., & Cho, K.-R. (2006). Modeling and analysis of the system bus latency on the SoC platform. En Proceedings of the 2006 International Workshop on System Level Interconnect Prediction (pp. 67–74). Association for Computing Machinery. https://doi.org/10.1145/1117278.1117293

Criss, K., Bains, K., Agarwal, R., Bennett, T., Grunzke, T., Kim, J. K., Chung, H., & Jang, M. (2020). Improving memory reliability by bounding DRAM faults: DDR5 improved reliability features. En Proceedings of the International Symposium on Memory Systems (pp. 317–322). Association for Computing Machinery. https://doi.org/10.1145/3422575.3422803

Dai, F., Hossain, M. A., & Wang, Y. (2025). State of the art in parallel and distributed systems: Emerging trends and challenges. Electronics, 14(4), Article 677. https://doi.org/10.3390/electronics14040677

Dally, W. J. (1990). Performance analysis of k-ary n-cube interconnection networks. IEEE Transactions on Computers, 39(6), 775–785.

Dominguez Perez, D. (2016). Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures [Presentación]. Iowa State University.

Durán, J. M. (2013). Explaining simulated phenomena: A defense of the epistemic power of computer simulations [Tesis doctoral, Universität Stuttgart].

Farshin, A., Roozbeh, A., Maguire, G. Q., Jr., & Kostić, D. (2020). Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks. En 2020 USENIX Annual Technical Conference (pp. 673–689). USENIX Association.

Ferreira, L. C. B. C., Borchardt, A. da R., Cardoso, G. dos S., Lemes, D. A. M., Sousa, G. R. dos R. de, Bauer Neto, F., Lima, E. R. de, Fraidenraich, G., Cardieri, P., & Meloni, L. G. P. (2022). Edge computing and microservices middleware for home energy management systems. IEEE Access, 10, 109663–109679. https://doi.org/10.1109/ACCESS.2022.3214229

Flynn, M. J. (1972). Some computer organizations and their effectiveness. IEEE Transactions on Computers, C-21(9), 948–960. https://doi.org/10.1109/TC.1972.5009071

Gunarathne, T., Wu, T. L., Qiu, J., & Fox, G. (2010). Cloud computing paradigms for pleasingly parallel biomedical applications. En Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (pp. 460–469). Association for Computing Machinery. https://doi.org/10.1145/1851476.1851544

Gupta, A. K., & Dally, W. J. (2005). Topology optimization of interconnection networks. Computer Systems Laboratory, Stanford University.

Harjula, E., Artemenko, A., & Forsström, S. (2022). Edge computing for industrial IoT: Challenges and solutions [Book chapter].

Heinikoski, J. (2023). Microprogrammed machine simulation using React [Bachelor's thesis, Lappeenranta–Lahti University of Technology LUT].

Jackson, C. R. (2007). Investigating serial microprocessors for FPGAs [MEng dissertation, University of York].

Jacobson, H., & Gopalakrishnan, G. (2001). Application-specific programmable control for high performance asynchronous circuits. Proceedings of IEEE Special Issue on Asynchronous Circuits & Systems, 1–12.

Jesshope, C., & Luo, B. (2000). Micro-threading: A new approach to future RISC. Institute of Information Sciences and Technology, Massey University.

JIS College of Engineering. (2015). Full curriculum and syllabus of B.Tech. in Computer Science and Engineering: Programme under autonomous framework, 1st to 8th semester. Department of Computer Science and Engineering, JIS College of Engineering.

Kalidas, R., Daga, M., Krommydas, K., & Feng, W.-C. (2015). On the performance, energy, and power of data-access methods in heterogeneous computing systems. En 11th Workshop on High-Performance, Power-Aware Computing. Hyderabad, India.

Kashyap, A., & Lu, X. (2022). NVMe-oAF: Towards adaptive NVMe-oF for IO-intensive workloads on HPC cloud. En Proceedings of the 31st

International Symposium on High-Performance Parallel and Distributed Computing (pp. 56–70). Association for Computing Machinery. https://doi.org/10.1145/3502181.3531476

Kozar, A., von Bleichert, J., Breß, S., Grulich, P. M., Lutz, C., Rabl, T., Rosenfeld, V., Traub, J., Zeuch, S., & Markl, V. (2025). Query processing on heterogeneous hardware. En K.-U. Sattler et al. (Eds.), Scalable data management for future hardware (pp. 39–64). Springer. https://doi.org/10.1007/978-3-031-74097-8_2

Kwon, T.-J., Sondeen, J., & Draper, J. (2005). Design trade-offs in floating-point unit implementation for embedded and processing-in-memory systems. En Proceedings of the IEEE International Symposium on Circuits and Systems (pp. 3331–3334). IEEE.

Lee, J.-E., Choi, K., & Dutt, N. D. (2003). Energy-efficient instruction set synthesis for application-specific processors. En Proceedings of the 2003 International Symposium on Low Power Electronics and Design (pp. 330–333). Association for Computing Machinery.

Li, C., Xue, Y., Wang, J., Zhang, W., & Li, T. (2018). Edge-oriented computing paradigms: A survey on architecture design and system management. ACM Computing Surveys, 51(2), Article 39. https://doi.org/10.1145/3154815

Liang, J., Tessier, R., & Mencer, O. (2003). Floating point unit generation and evaluation for FPGAs. En Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. IEEE.

Mazumdar, S., Seybold, D., Kritikos, K., & Verginadis, Y. (2019). A survey on data storage and placement methodologies for Cloud-Big Data ecosystem. Journal of Big Data, 6, Article 15. https://doi.org/10.1186/s40537-019-0178-3

Nakao, M., Sakai, M., Hanada, Y., Murai, H., & Sato, M. (2021). Graph optimization algorithm for low-latency interconnection networks. Parallel

Computing, 106, Article 102805. https://doi.org/10.1016/j.parco.2021.102805

Ngoko, Y., & Trystram, D. (2018). Revisiting Flynn's classification: The portfolio approach. En Euro-Par 2017: Parallel Processing Workshops (pp. 227–239). Springer. https://doi.org/10.1007/978-3-319-75178-8_19

Nikolić, G., Dimitrijević, B., Nikolić, T., & Stojčev, M. (2022). Fifty years of microprocessor evolution: From single CPU to multicore and manycore systems. Facta Universitatis, Series: Electronics and Energetics, 35(2), 155–186. https://doi.org/10.2298/FUEE2202155N

Nordholz, J. (2020). Design of a symbolically executable embedded hypervisor. En Proceedings of the Fifteenth European Conference on Computer Systems. Association for Computing Machinery. https://doi.org/10.1145/3342195.3387516

Oladele, O. K. (2024). Neuromorphic computing and spiking neural networks: Bridging neuroscience with AI hardware [Manuscript].

Özyılmaz, M. M. (2026). A comparative analysis of ARM and x86-64 laptop-class processors: Architecture, assembly-level performance, and energy efficiency. arXiv. https://arxiv.org/abs/2604.18896

Patel, M., Kim, J. S., Hassan, H., & Mutlu, O. (2019). Understanding and modeling on-die error correction in modern DRAM: An experimental study using real devices. En Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE.

Păun, G., Rozenberg, G., & Salomaa, A. (2003). ДНК-компьютер. Новая парадигма вычислений [DNA computing: New computing paradigms] (D. S. Ananichev, I. S. Kiseleva, & O. B. Finogenova, Trads.; M. V. Volkov, Ed.). Mir. (Obra original publicada en 1998)

Pizzol, G. D., Pilla, M. L., & Navaux, P. O. A. (s. f.). Branch prediction x performance: An analysis on superscalar processors. Universidade Federal do Rio Grande do Sul.

Ponrani, M. A., Thanishtha, J., & Swasthika, M. (2026). Design and analysis of 5 stage pipelined CPU with hazard handling. ITM Web of Conferences, 82, Article 01005. https://doi.org/10.1051/itmconf/20268201005

Proctor, F. M., & Shackleford, W. P. (s. f.). Real-time operating system timing jitter and its impact on motor control. National Institute of Standards and Technology.

Proietti, R., Cao, Z., Nitta, C. J., Li, Y., & Yoo, S. J. B. (2015). A scalable, low-latency, high-throughput, optical interconnect architecture based on arrayed waveguide grating routers. Journal of Lightwave Technology, 33(4), 911–920. https://doi.org/10.1109/JLT.2015.2395352

Qin, J. (2025). Design of big data management system based on distributed architecture. En Proceedings of the 2025 5th International Conference on Big Data, Artificial Intelligence and Risk Management (pp. 618–623). Association for Computing Machinery. https://doi.org/10.1145/3800227.3800318

Rosdiyanto, R., Zuhro, S. F., & Nisfuwadi, R. (2025). Comparative analysis of RISC and CISC architectures in modern embedded system development. Bit-Tech, 8(2), 1910–1917. https://doi.org/10.32877/bt.v8i2.3161

Sá, B., Valente, L., Martins, J., Rossi, D., Benini, L., & Pinto, S. (2023). CVA6 RISC-V virtualization: Architecture, microarchitecture, and design space exploration. arXiv. https://arxiv.org/abs/2302.02969

Sehgal, N. K., Bhatt, P. C. P., & Acken, J. M. (2023). Cloud computing with security and scalability: Concepts and practices (3rd ed.). Springer. https://doi.org/10.1007/978-3-031-07242-0

Shacham, A., & Bergman, K. (2007). Building ultralow-latency interconnection networks using photonic integration. IEEE Micro, 27(4), 6–20.

Sharan, H., Vutukuru, M., & Panda, B. (2023). DDIOSim: A microarchitecture simulator for Data Direct I/O technology [Conference paper].

Shawish, A., & Salama, M. (2014). Cloud computing: Paradigms and technologies. En F. Xhafa & N. Bessis (Eds.), Inter-cooperative collective intelligence: Techniques and applications (pp. 39–67). Springer. https://doi.org/10.1007/978-3-642-35016-0_2

Smolyar, I., Markuze, A., Pismenny, B., Eran, H., Zellweger, G., Bolen, A., Liss, L., Morrison, A., & Tsafrir, D. (2020). IOctopus: Outsmarting nonuniform DMA. En Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 101–115). Association for Computing Machinery. https://doi.org/10.1145/3373376.3378509

Stanford University. (s. f.). Flynn's taxonomy [Diapositivas de clase].

Ştefan, G. (2006). Integral parallel computation. Proceedings of the Romanian Academy, Series A, 7(3), 1–8.

Sultanpuri, H. R. (2025). Implementation of RISC-V P-extension ALU [Master's thesis, Aalto University].

Thinh, P. V., & Tung, N. T. (2026). From mechanical computation to algorithmic intelligence: An evolutionary epistemological analysis of computer science. European Journal of Applied Science, Engineering and Technology, 4(2), 312–320. https://doi.org/10.59324/ejaset.2026.4(2).20

Vaithianathan, M. (2025). Memory hierarchy optimization strategies for high-performance computing architectures. International Journal of Emerging Trends in Computer Science and Information Technology, 6(1), 23–34. https://doi.org/10.63282/3050-9246/IJETCSIT-V6I1P103

Vallverdú i Segura, J. (2009). Computational epistemology and e-science: A new way of thinking. Minds and Machines, 19(4), 557–567. https://doi.org/10.1007/s11023-009-9168-0

Van der Steen, A. J. (2008). Overview of recent supercomputers. NCF/HPC Research.

Vogginger, B., Rostami, A., Jain, V., Arfa, S., Hantsch, A., Kappel, D., Schäfer, M., Faltings, U., Gonzalez, H. A., Liu, C., Mayr, C., & Maaß, W. (2024). Neuromorphic hardware for sustainable AI data centers. arXiv. https://arxiv.org/abs/2402.02521

Von Neumann, J. (1945). First draft of a report on the EDVAC. University of Pennsylvania.

Yellapragada, S. (2022). Analysis of communication and computation overlap in accelerated programs [Tesis de maestría, University of Illinois Urbana-Champaign].

Zhang, H., Chen, G., Ooi, B. C., Tan, K.-L., & Zhang, M. (2015). In-memory big data management and processing: A survey. IEEE Transactions on Knowledge and Data Engineering, 27(7), 1920–1948. https://doi.org/10.1109/TKDE.2015.2427795

Zhang, J., Shen, D., Chen, D., Ming, D., Ren, D., & Diao, Z. (2024). ISMSFuse: Multi-modal fusing recognition algorithm for rice bacterial blight disease adaptable in edge computing scenarios. Computers and Electronics in Agriculture, 223, Article 109089. https://doi.org/10.1016/j.compag.2024.109089

Zhou, P., Önder, S., & Carr, S. (2005). Fast branch misprediction recovery in out-of-order superscalar processors. En Proceedings of the 19th Annual International Conference on Supercomputing (pp. 41–50). Association for Computing Machinery.

Zhou, Y., Zhang, D., & Xiong, N. (2017). Post-cloud computing paradigms: A survey and comparison. Tsinghua Science and Technology, 22(6), 714–732. https://doi.org/10.23919/TST.2017.8195353

Zhu, J., Wang, L., Xiao, L., & Qin, G. (2023). uDMA: An efficient user-level DMA for NVMe SSDs. Applied Sciences, 13(2), Article 960. https://doi.org/10.3390/app13020960

Arquitectura y Organización de Computadores

De los Sistemas Legados a la Computación de Alto Rendimiento

Este libro ofrece una visión rigurosa, actualizada y formativa sobre la arquitectura y organización de computadores, desde sus fundamentos históricos y epistemológicos hasta los paradigmas de computación de alto rendimiento. A través de un enfoque académico, integra el estudio de las familias de computadores, jerarquías de memoria, microprocesadores modernos, entrada/salida, DMA, virtualización, nube, edge computing y sostenibilidad tecnológica. La obra permite comprender cómo los sistemas computacionales procesan, almacenan, transfieren y protegen datos en entornos cada vez más complejos, distribuidos y heterogéneos, constituyéndose en un recurso valioso para estudiantes, docentes, investigadores y profesionales del área informática.

ISBN: 978-9907-9519-9-8

Mgs. César Andrés Mero Baquerizo

Mgs. Betty Azucena Macas Padilla

Mgs. Juan Carlos Vasco Delgado

EDUINCA
EDITORIAL DE EDUCACIÓN, INVESTIGACIÓN Y CULTURA ACADÉMICA