Tema 16: Procesamiento en paralelo

Arquitectura de Computadoras

Ing. Nicolás Majorel Padilla (npadilla@herrera.unt.edu.ar)

http://microprocesadores.unt.edu.ar/arqcom/

Temas que veremos

- Definición, ventajas y desventajas.
- Performance en sistemas paralelos.
  - Limitaciones, Eficiencia, Escalabilidad.
- Taxonomía de Flynn.
- Paralelismo a nivel Datos: SIMD.
- Paralelismo a nivel *Threads*: *Multithreading*.
- Clasificación según el acceso a memoria.
  - SMP (UMA o NUMA) y Clusters.
- Coherencia en SMP.
- Paso de Mensajes en Clusters.
- Arquitecturas de Dominio Específico

Lectura recomendada

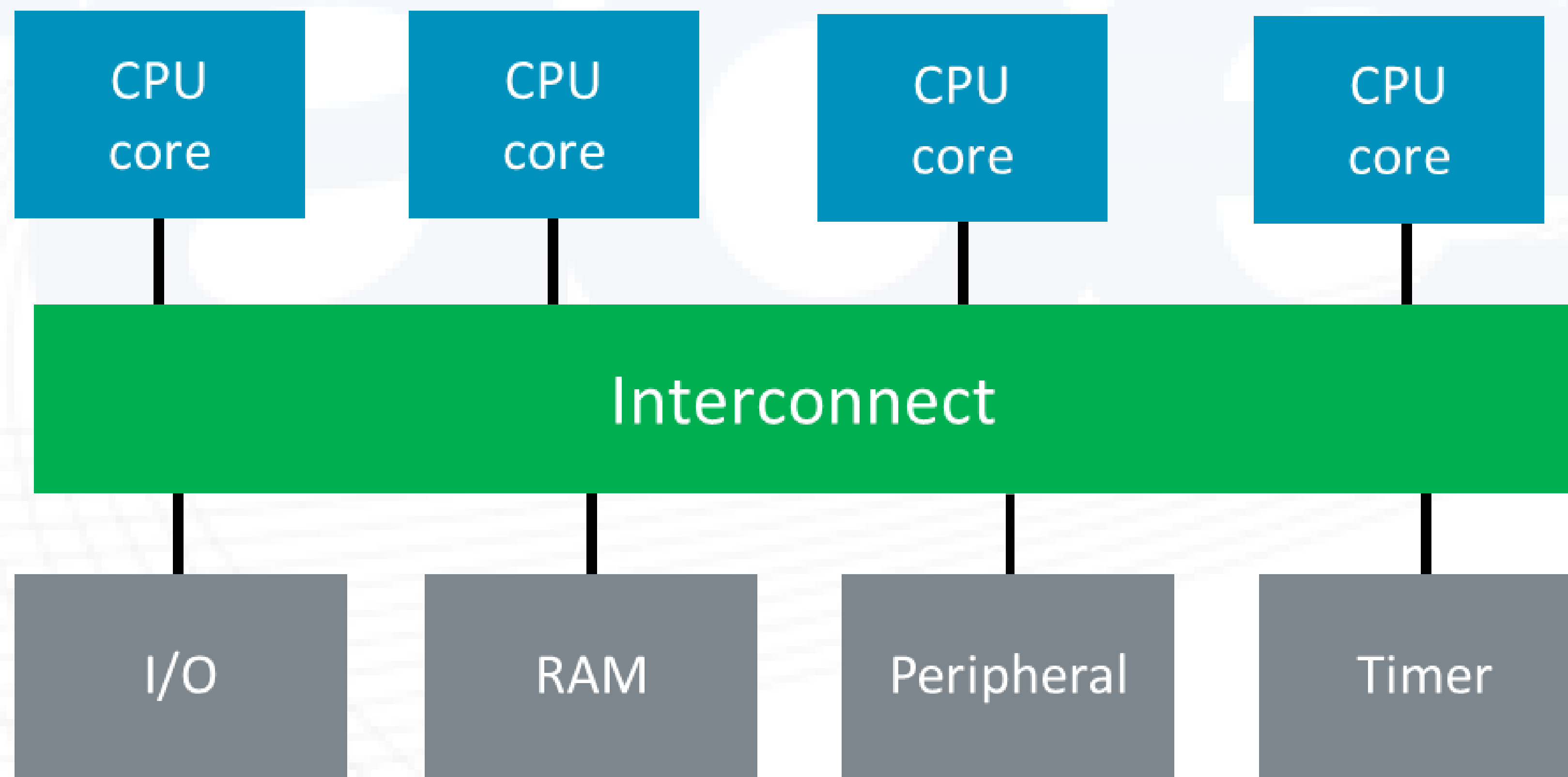
- Computer Organization and Design, RISC-V Edition (2da ed, 2021):
  - Sección 6.1: *Introduction*
  - Sección 6.2: *The Difficulty of Creating Parallel Processing Programs*
  - Sección 6.3: *SISD, MIMD, SIMD, SPMD and Vector*
  - Sección 6.4: *Hardware Multithreading*
  - Sección 6.5: *Multicore and Other Shared Memory Multiprocessors*
  - Sección 5.10: *Paralellism and Memory Hierarchy: Cache Coherence*
  - Sección 6.8: *Clusters, Warehouse Scale Computers, and Other Message-Passing Multiprocessors*
  - Sección 6.7: *Domain-specific Architectures*
  - Sección 6.14: *Fallacies and Pitfalls*
  - Sección 6.15: *Concluding Remarks*

¿Dónde estamos parados?

- Vimos que ILP era una manera de incrementar la performance.
  - Además, es bastante transparente para los programadores.
  - Pero estaba fuertemente limitado por las dependencias, en códigos mayormente secuenciales.
- Vimos que la *Memory Wall* empezó a convertirse en un problema grande.
  - E intentamos reducirla con una jerarquía de memorias caché.
- Vimos que la *Power Wall* y el fin del escalamiento de Dennard nos impiden seguir aumentando la frecuencia para mejorar la performance.
  - Y nos generan un cambio de paradigma.
  - **Buscar mejorar la performance mediante el uso de varios procesadores juntos (*multicore*).**

Multiprocesamiento

- En realidad, como concepto ya existía desde antes.
  - Todos los factores anteriores lo impulsaron, popularizaron y lo llevaron a nuestros bolsillos.
  - Se pasó de conectar varios procesadores en un sistema, a conectar varios núcleos en un mismo procesador.
  - El concepto de ***System on a Chip*** **(SoC)** que vimos en temas anteriores.



Multiprocesamiento: Ventajas y desventajas

- Más procesadores…
  - Aumentan la performance (en principio).
  - Disminuyen el consumo de energía (en principio).
  - Aumentan la disponibilidad.
  - Disminuyen la eficiencia.
  - Generan problemas de sincronización y de coherencia de datos.
  - Generan dificultad para los programadores (no son transparentes).

Implicancias de multiprocesamiento

- Cuatro estudiantes quieren hacer un TP de esta materia en paralelo, para hacerlo más rápido.
  - Deberían **dividir** el TP en partes iguales (**balance de carga**).
    - Si no, algunos terminarán antes que los demás y tendrán que esperar (**sincronización**).
    - Asumimos que es posible dividir el TP en partes iguales, lo cual no es siempre válido.
  - Asignar una parte del TP a cada uno de ellos (***scheduling***).
  - Probablemente pierdan tiempo conversando entre ellos, en vez de hacer su parte del TP (***overhead* por comunicación**).
- Si los estudiantes fueran ocho, las implicancias se agravan.

- Todos los tipos de paralelismo que veremos tendrán que manejar estos detalles.

Consideraciones sobre performance

- Un sistema es **escalable** si al incrementar los recursos (Hw), se logra una mejora proporcional en la performance.
  - Si realizamos una tarea con *p* procesadores, *¿mejorará p veces?*
  - **A (*p*) = t_ej(1 procesador) / t_ej(*p* procesadores)**
  - Lo ideal sería obtener una aceleración **lineal**.

Consideraciones sobre performance

- Se desea sumar 8000 números independientes, cada suma demora 1 T, se cuenta con 8 procesadores.
- Separamos la tarea en pedazos independientes.
  - Cada procesador suma 1000 números.
  - Son necesarios 3 ciclos más para sumar los subtotales (coordinación).
- A(8) = 8000 / 1003 = 7,98
  - Aceleración casi ideal. Excelente caso.
  - No estamos considerando problemas de sincronización.
    - Que un procesador quede esperando datos de otro.

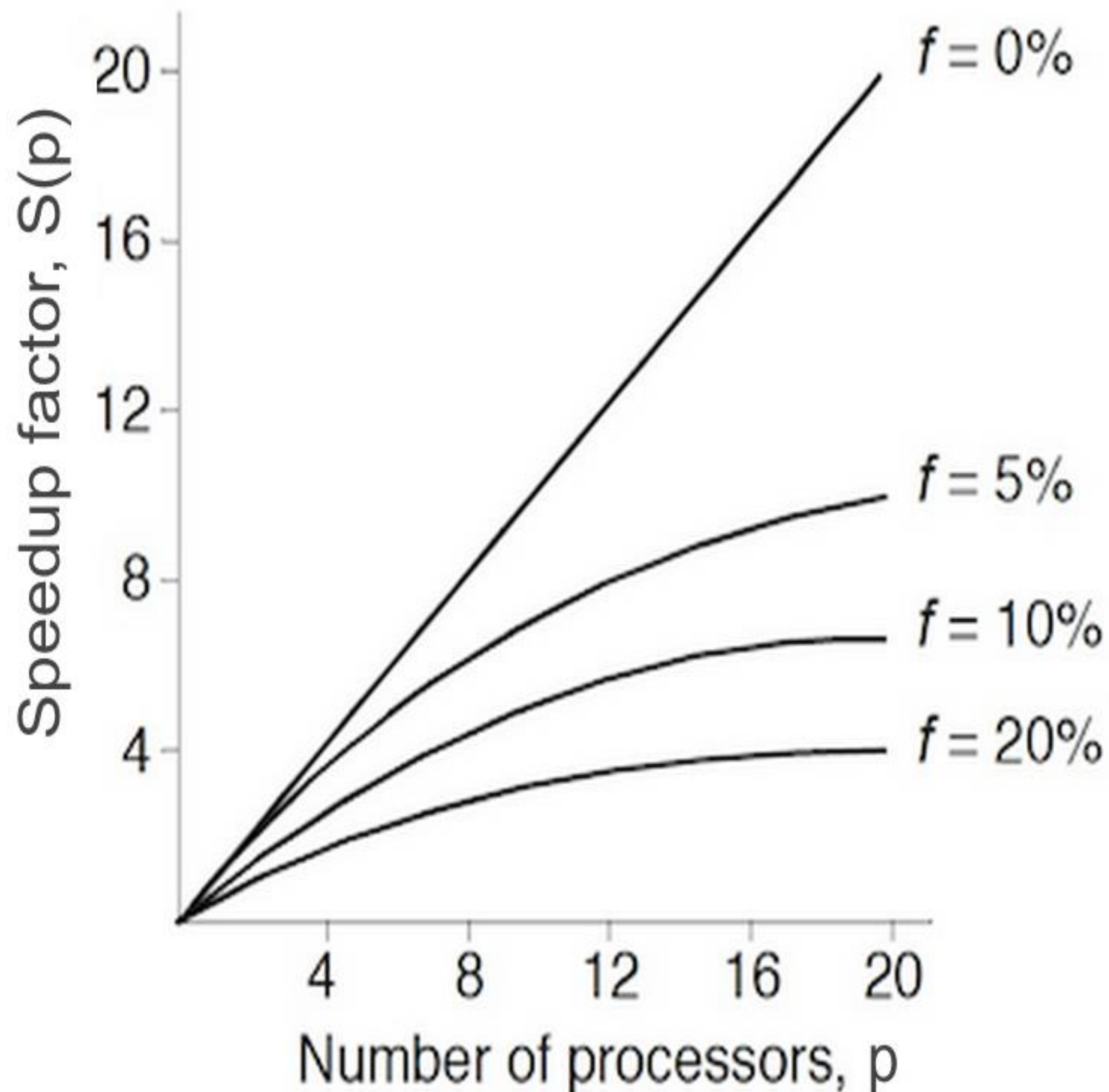
Consideraciones sobre performance

- El ejemplo anterior es bastante irreal.
  - Puede haber limitaciones por el uso de memoria, por el uso de I/O, por el medio de conexión.
- Ley de Amdahl, otra vez.
  - La mejora total de un sistema está limitada por la parte no modificada.
  - $\mathbf{A\ (\mathit{p}) = 1\ /\ [1 - F + (F\ /\ \mathit{p})]}$

Consideraciones sobre performance

- Tenemos los mismos 8 procesadores anteriores, y queremos ejecutar una tarea que tiene un 20% de código secuencial, y el 80% restante es completamente paralelizable.
  - F = 0,8; p = 8.
  - A = 1 / [0,2 + (0,8 / 8)] = **3,33**
- Fuerte baja. Ni siquiera la mitad del ideal.
  - Y nada que ver con el Hw. Limitación exclusiva de Sw.
- El mayor desafío del procesamiento paralelo es decidir cómo es mejor partir un problema en pedazos que puedan ser procesados por separado.

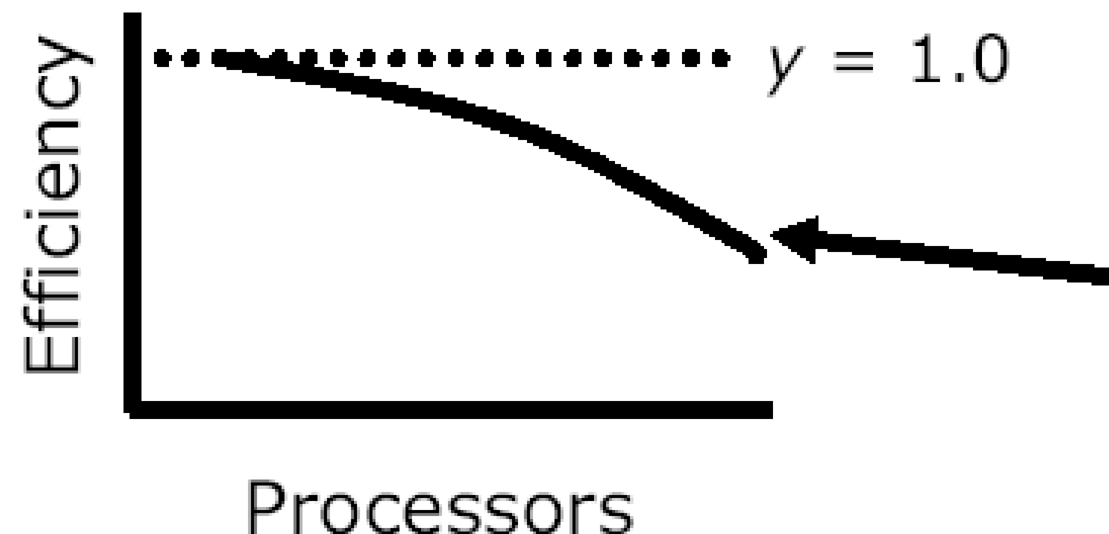
Conclusiones sobre Ley de Amdahl



- La fracción serial impone una fuerte limitación al paralelismo.
- Con muchos procesadores empieza a disminuir la aceleración.
  - Debido a problemas de comunicación y/o coordinación (congestión).
- Asume que el problema a resolver es de tamaño fijo.
  - La carga de trabajo de cada procesador va disminuyendo.
- Esto se conoce como **Escalabilidad fuerte** (*Strong Scaling*)

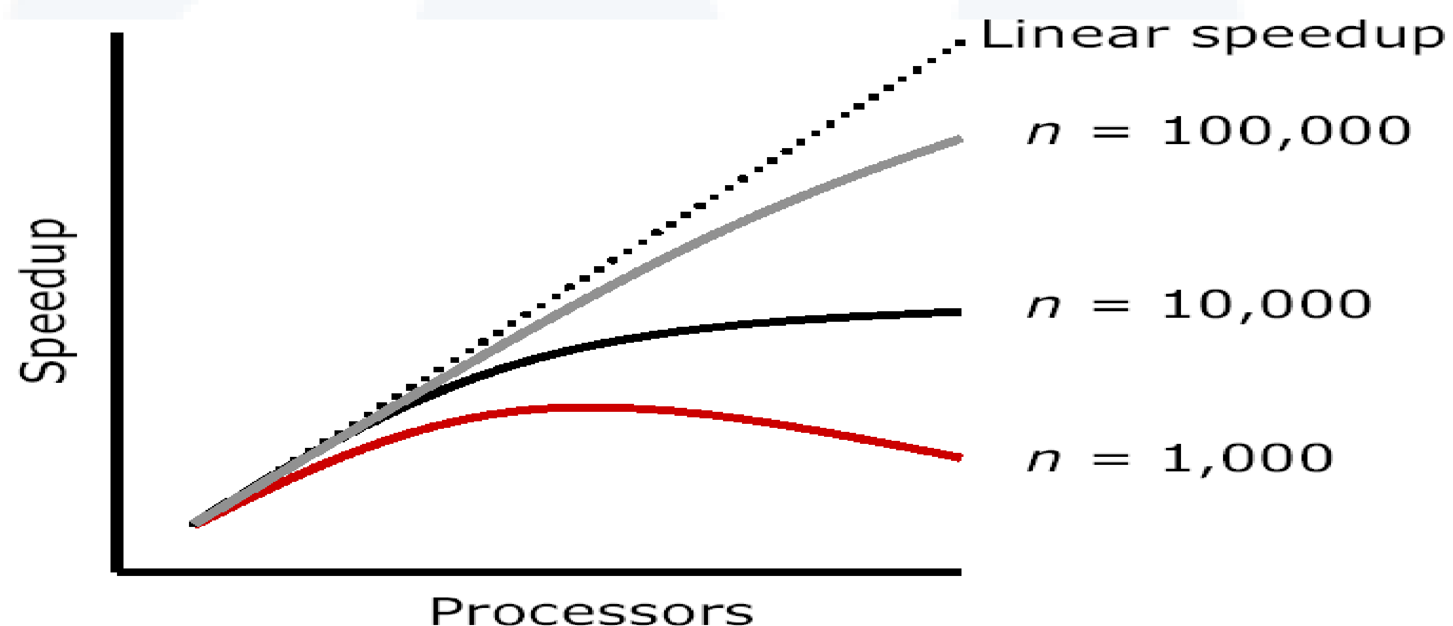
Eficiencia del paralelismo

- La aceleración no es la única métrica importante.
  - Ej: obtener una aceleración 2 usando 10 procesadores, ¿es un buen resultado?
- La eficiencia es cuan bien se utilizan los recursos disponibles.
  - **E (*p*) = A (*p*) / *p***
  - Ej: si con 8 procesadores se obtiene una aceleración de 3,33, la eficiencia será del 41,6%.



Alternativa para mejorar Escalabilidad

- Aumentar los procesadores para realizar una tarea baja eficiencia, y está limitado por la parte serial.
- Entonces… les demos muchas tareas.
  - **Adaptar la cantidad de trabajo al paralelismo disponible.**
  - Cuanto mayor sea el conjunto de datos, más próxima será la aceleración al ideal.
    - La carga de trabajo de cada procesador se mantiene constante.
  - Esto se conoce como **Escalabilidad débil** (*Weak Scaling*).
  - No siempre es posible. Depende del perfil y del tamaño de la tarea.

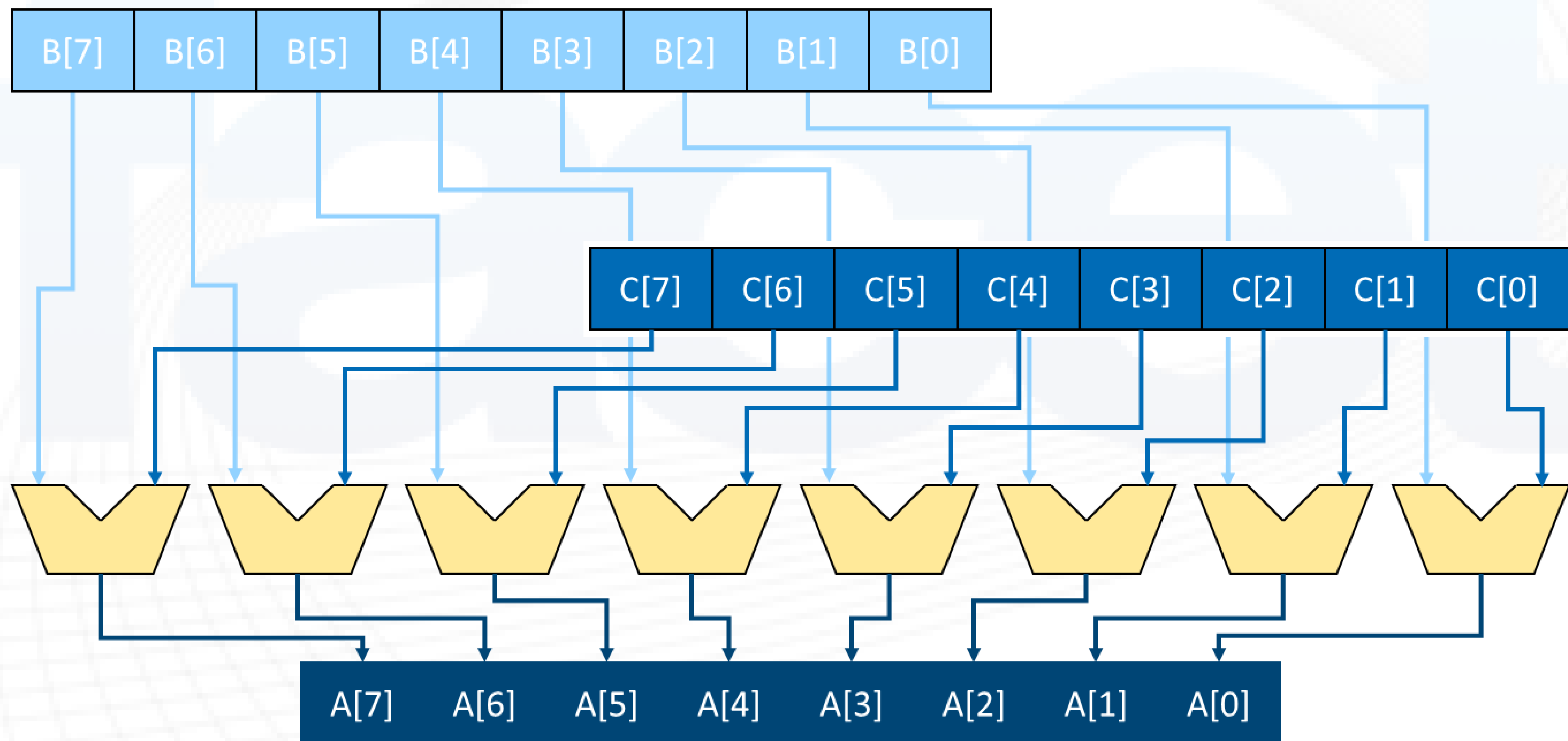


Una clasificación: Taxonomía de Flynn

- Hecha alrededor de 1960, propone categorías según la combinación entre cantidad de instrucciones y cantidad de datos.
- SISD: Single Instruction, Single Data.
  - Un procesador convencional que vimos hasta aquí. Todos los procesadores previos a 2004.
- **SIMD:** Single Instruction, Multiple Data.
  - Una única instrucción opera sobre un conjunto de datos.
- MISD: Multiple Instruction, Single Data.
  - No utilizado.
- **MIMD:** Multiple Instruction, Multiple Data.
  - Múltiples porciones de un mismo programa se ejecutan en distintos procesadores, con distintos datos. Un procesador estándar desde 2004.

SIMD

- Operan elemento a elemento sobre un vector de datos.
  - Ejecutan una suma de dos vectores de varios elementos en un solo ciclo.
  - Ideales para trabajar con arrays en lazos for.
- Bastante simples. Sincronizados.



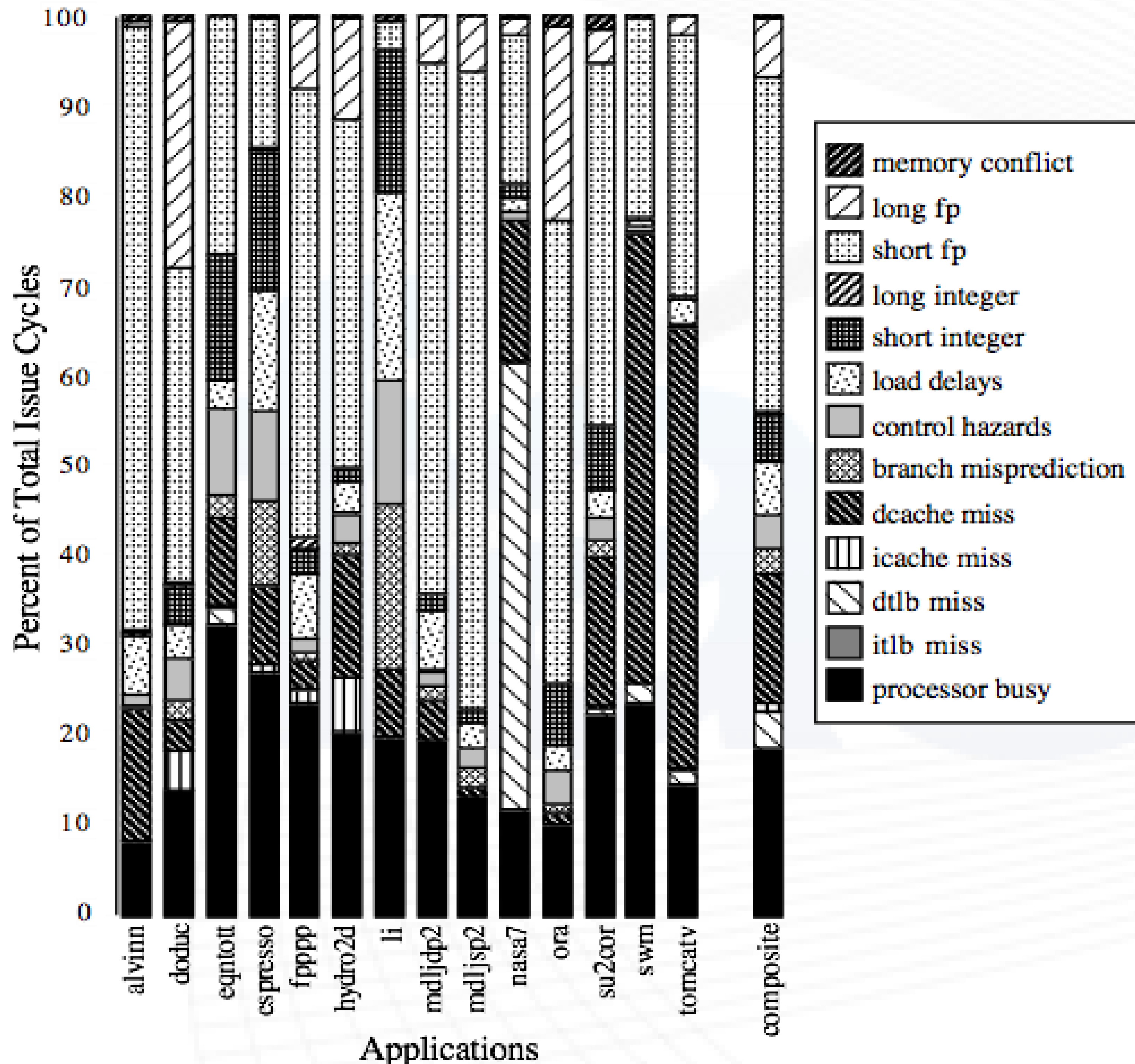
SIMD

- La longitud de los registros es fija, **definida por el ISA**.
  - Nuevas longitudes implica agregar nuevas instrucciones.
    - Implica soporte en los compiladores, SO, etc. ¡Y reescribir/recompilar las aplicaciones!
    - Es un problema para los ISA de formatos de longitud fija.
- Muy popular en la arquitectura x86 de Intel.
  - Extensiones MMX (64 bits), SSE (128 bits), SSE2, AVX (256 bits), AVX2, AVX512 (512 bits).
  - En las arquitecturas ARM, existe la extensión NEON.
  - En RISC-V, hay una extensión no estándar (P).
- Mercado principal: aplicaciones multimedia.
  - Video maneja RGB: 3x8 bits + 8 bits de transparencia.
  - Audio: 24 bits/muestra.
- Explotan el **paralelismo a nivel datos (DLP)**.

SIMD – Variantes

- Dentro de esta categoría podrían entrar también arquitecturas vectoriales y los procesadores gráficos (GPU).
  - Distintos en esencia, pero también explotan el DLP.
- El ISA de RISC-V provee una extensión especial (V) para operaciones vectoriales.
  - Imitada por ARM en sus ISAs más reciente, llamada SVE (*Scalable Vector Extension*).
  - No posee registros de longitud fija.
  - Permite varias implementaciones diferentes de Hw, pero simplificando el Sw.
    - No se modifica el ISA, ni las aplicaciones, ni los compiladores, SO, etc.
- Estos temas irían en "Arquitectura 2" ☺

Una variante de MIMD: Motivación



- Un análisis de ILP para un procesador superescalar de 8 vías mostró que la mayoría del tiempo, el procesador está parado sin hacer nada.
- No se puede cambiar de tarea.
- Surgió el concepto de hilos de ejecución (*threads*).

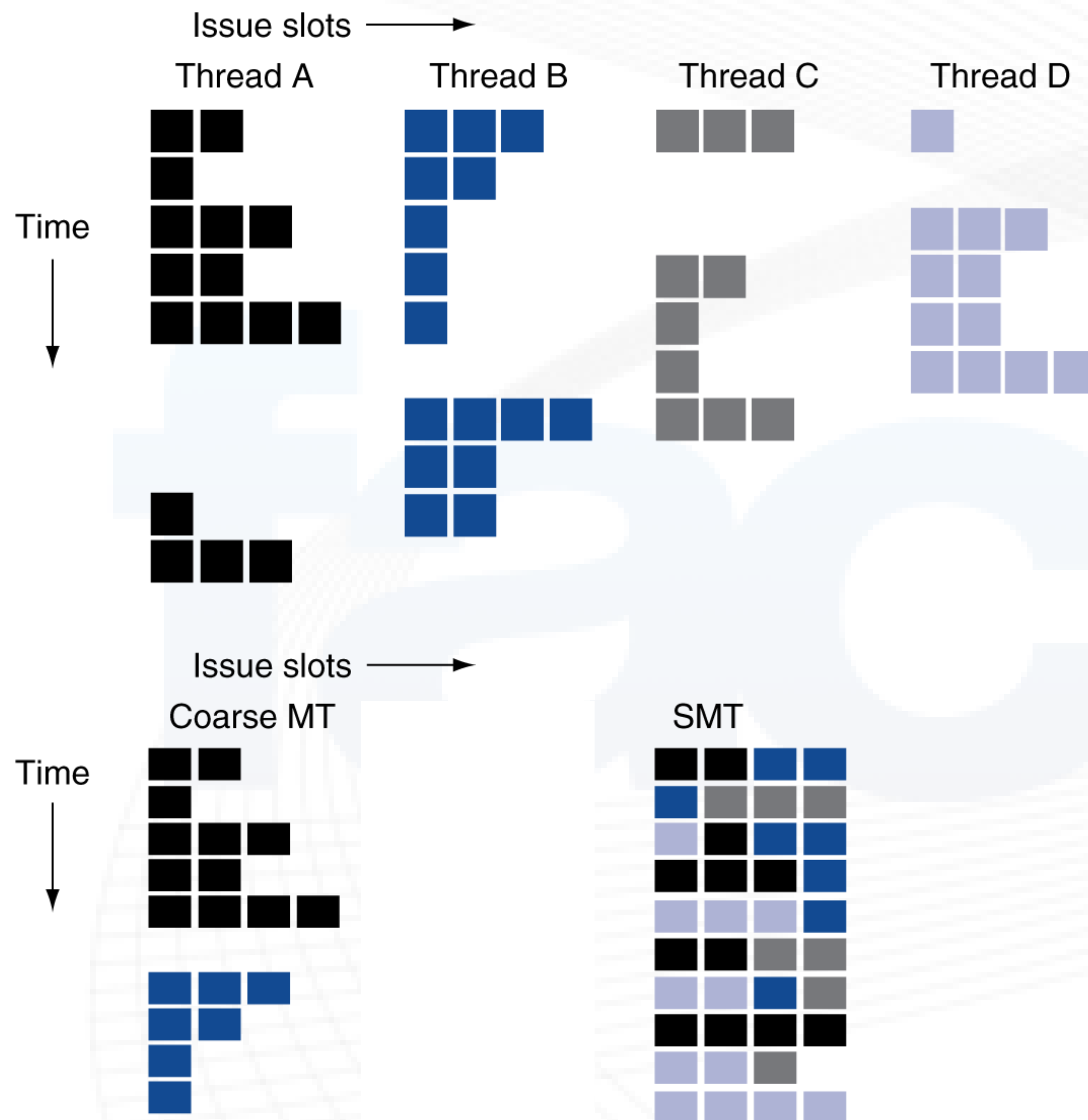
Threads: definiciones

- Los hilos de ejecución son funciones **independientes** de un mismo programa.
  - Comparten los recursos del programa.
  - Cada *thread* tiene su propio PC y su propio estado (banco de registros, puntero a la base de la tabla de páginas, etc).
  - Es mucho más rápido cambiar de *thread* que de proceso.
    - El cambio suele ser instantáneo.
- Ejemplos:
  - Sonido estéreo.
  - Corrector ortográfico de MS Word.
  - Ejemplo de I/O visto en SMM.
- Si las instrucciones provienen de distintos *threads* de un programa, ya se sabe que son independientes.

Una alternativa a MIMD: *Multithreading*

- Un único procesador que soporta múltiples *threads* de ejecución.
  - Cada *thread* es considerado como un CPU virtual.
- Cuando la ejecución de un *thread* se ve detenida por alguna parada (p.ej, un fallo de caché), se cambia a otro *thread*.
  - La parada queda enmascarada, la ejecución continúa (pero de otro *thread*).
  - Esto se conoce como *coarse-grained multithreading*.
  - La clave es que el cambio de thread sea más rápido que la resolución de la parada.
- ***Multithreading* simultáneo (SMT)** aprovecha las características de un procesador superescalar con emisión dinámica múltiple de instrucciones, y busca emitir siempre instrucciones de diferentes *threads* para llenar las unidades funcionales disponibles.

Comparación *coarse-grained* MT vs SMT



- Cuatro threads diferentes, que se ejecutan en un procesador superescalar con emisión múltiple de 4 instrucciones.
- Las dependencias hacen que no siempre se puedan emitir 4 instrucciones en cada ciclo (ILP).
- SMT aprovecha que las instrucciones de distintos threads no tienen dependencias entre sí.

Multithreading Simultáneo – Ventajas

- Es una forma de explotar el **paralelismo a nivel *threads* (TLP)**.
- Caso de uso ideal: un *thread* principal más un *thread* pequeño que ejecute en paralelo, y cuando haya mucha comunicación entre ambos.
- 2 *threads* en SMT generan una mejora promedio del 20-25%
  - Con muy poco agregado de recursos: duplicar el PC, el banco de registros, y mantener tablas de páginas separadas.
    - Aproximadamente un aumento del 5% del área.
  - En *threads* con bajo ILP, es posible obtener una aceleración casi lineal.

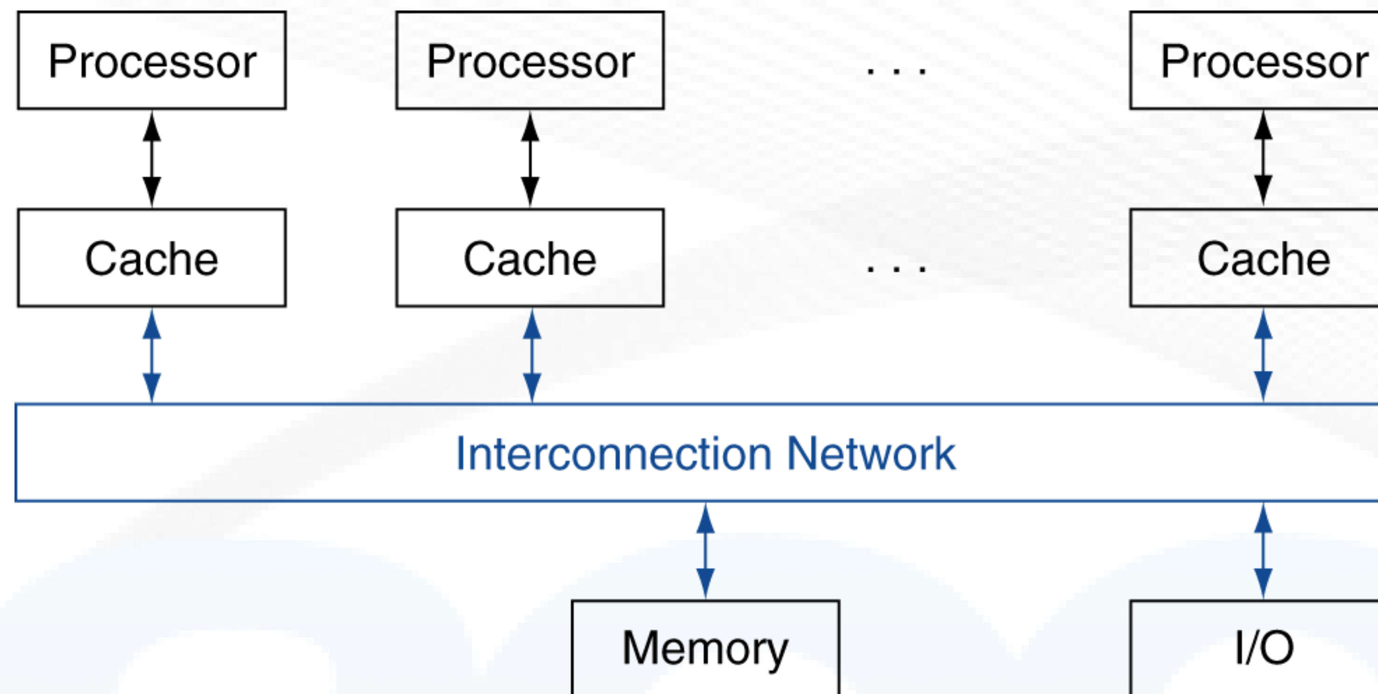
Multithreading Simultáneo – Desventajas

- Limitación 1: **software**
  - Procesadores con muchos *threads* por núcleo no mostraron una mejora que valga la pena.
  - Los programas no usan tantos *threads*.
  - La ejecución de programas de un único *thread* puede empeorar.
- Conclusión: **varios núcleos, pocos *threads* por núcleo.**
- Limitación 2: **seguridad**
  - Se descubrió que el soporte para SMT representa una seria vulnerabilidad de seguridad.
  - Porque implica compartir muchos recursos: cachés de datos e instrucciones, TLB de datos e instrucciones, buffers de escrituras, buffers de predicción de saltos, etc.
  - Más detalles en Tema 17.

Otra clasificación de multiprocesamiento

- Se pueden clasificar los sistemas de multiprocesamiento según si su acceso a memoria es compartido o no.
- En un **acceso compartido a memoria**, todos los procesadores comparten un único espacio de direcciones físicas.
  - Todos los datos de memoria están disponibles para todos los procesadores en todo momento.
  - Mucho más simple de programar: a través de Ld/St.
  - Se necesitan operaciones atómicas en memoria para sincronización.
    - Operaciones que no se pueden interrumpir.
    - Fundamentales para Sistemas Operativos.
    - En el ISA de RISC-V, están incluidas en la extensión A. *¿Se acuerdan?*
- En un **acceso no compartido a memoria**, los procesadores tienen cada uno un espacio de direccionamiento privado.
  - Es necesario un mecanismo explícito para compartir datos.

Acceso compartido a memoria



- Denominados **SMP (*Shared Memory multiProcessors*)**.
  - **Acceso Uniforme a Memoria (UMA):** misma latencia para todos.
    - La mayoría de los multicores que conocemos.
    - Comparten datos a través del LLC (actualmente, L3).
  - **Acceso No Uniforme a Memoria (NUMA)**.
    - Los procesadores poseen memoria local, y por lo tanto la latencia de memoria no es siempre la misma.

UMA vs. NUMA

- La popularidad de SMP tipo UMA es porque son más fáciles de programar.
- Sin embargo, en algunos casos un sistema NUMA puede proveer latencias menores.
  - AMD Ryzen Threadripper 2970WX (del 2018) posee 4 núcleos, pero sólo 2 de ellos tienen acceso directo a su memoria local.
    - Los otros dos lo hacen a través de la *Infinity Fabric*.
    - Permiten que una aplicación sea dinámicamente movida a uno de los procesadores con memoria si es que necesita mayor performance (*Dynamic Local Mode*).
- A partir de entonces, son cada vez más populares las conexiones tipo NUMA.

Problema de Coherencia de Cachés

- Como la memoria es compartida, es necesario que se provea un mecanismo para **mantener la coherencia de datos**.
  - Mantener la ilusión de que los datos son almacenados en una única gran memoria.
- Dos (o más) procesadores diferentes podrían acceder a la misma dirección de memoria compartida y obtener valores diferentes.
- Ejemplo con cachés write-through (write-back presentan un comportamiento similar):

Time step	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A stores 1 into X	1	0	1

- ***¿Qué sucede si luego CPU B quiere leer X?***
- ***Y si un CPU C quiere luego leer X, ¿qué valor debería leer?***

Problema de Coherencia de Cachés

- En el ejemplo anterior, en caso que el CPU B desee leer el valor de X, debe proveerse un mecanismo para que lea el valor correcto, desde memoria o desde donde sea.
- Estos mecanismos se denominan **protocolos de coherencia de caché**.
- Se basan en llevar un registro de cómo y cuándo es compartido cada bloque de caché.
  - Cada bloque mantiene información sobre su compartición junto con la de su estado.
- Los protocolos de coherencia más populares son de tipo ***snooping***.
  - Todos los cachés son accesibles desde un **único bus compartido**.
  - Las transacciones en ese bus se realizan de a una por vez.
  - Todos los cachés monitorean permanentemente ese bus compartido para saber si tienen una copia del bloque que se está solicitando.
    - Si detectan que en el bus hay una transacción sobre un bloque del cual tienen una copia, entonces realizan alguna acción.

Protocolo simple de Coherencia

- Cuando un procesador A escribe en una dirección de memoria X, se marcan como inválidas todas las copias de X que puedan existir en los cachés de los demás procesadores (***write invalidate***).
  - Esto implica que una escritura hace que un dato sea exclusivo de un procesador.
  - Genera nuevos fallos, porque puede que el procesador B haya leído antes el dato X, y al intentar leerlo nuevamente no lo encuentra.
    - Estos son los fallos por Coherencia (la cuarta ‘C’ que dejamos pendiente).

Protocolo simple de Coherencia

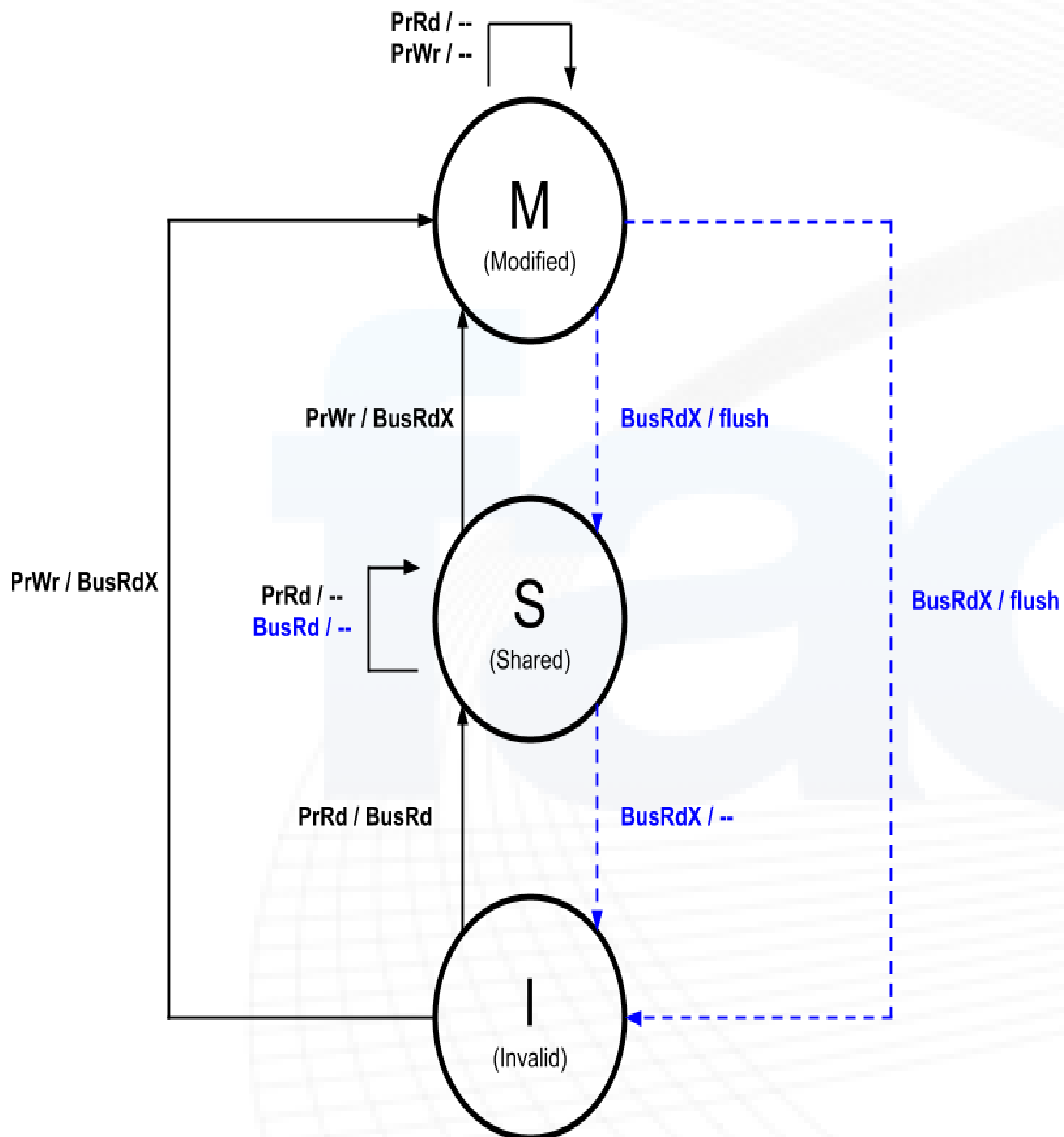
- Si un procesador B quiere leer una dirección de memoria X que es exclusiva de otro procesador A, se fuerza a que A haga un *write-back* a memoria de X y luego se produce la lectura.
  - En realidad, cuando A hace el *write-back*, al mismo tiempo le pasa el valor de X a B, porque el bus es compartido.
- Repasando el ejemplo visto anteriormente (ahora con cachés *write-back*):

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

Protocolo simple de Coherencia

- Estos protocolos son usualmente implementados como una MEF en cada LLC de cada procesador.
  - Estrictamente, en el límite entre memoria privada y memoria compartida.
  - Últimamente, **son implementados por las mismas interconexiones on-chip**.
- En la versión más simple, un bloque puede estar en uno de tres estados:
  - Inválido (I): el caché no posee este bloque.
  - Modificado (M): el bloque es exclusivo, sólo este caché tiene la versión válida.
  - Compartido (S): igual que en memoria, puede estar en otros procesadores.
- En una versión mejorada, se agrega un cuarto estado para evitar algunas transacciones por el bus: Exclusivo, pero no modificado.
  - El protocolo pasa a denominarse MESI, y es bastante popular.
- Hay más versiones, que agregan más estados.
  - P.ej: los procesadores Ryzen de AMD implementan un protocolo de 7 estados, MDOESFI.
- Y otros que no son *write-invalidate*, sino *write-update*.

MEF para Protocolo de Coherencia MSI

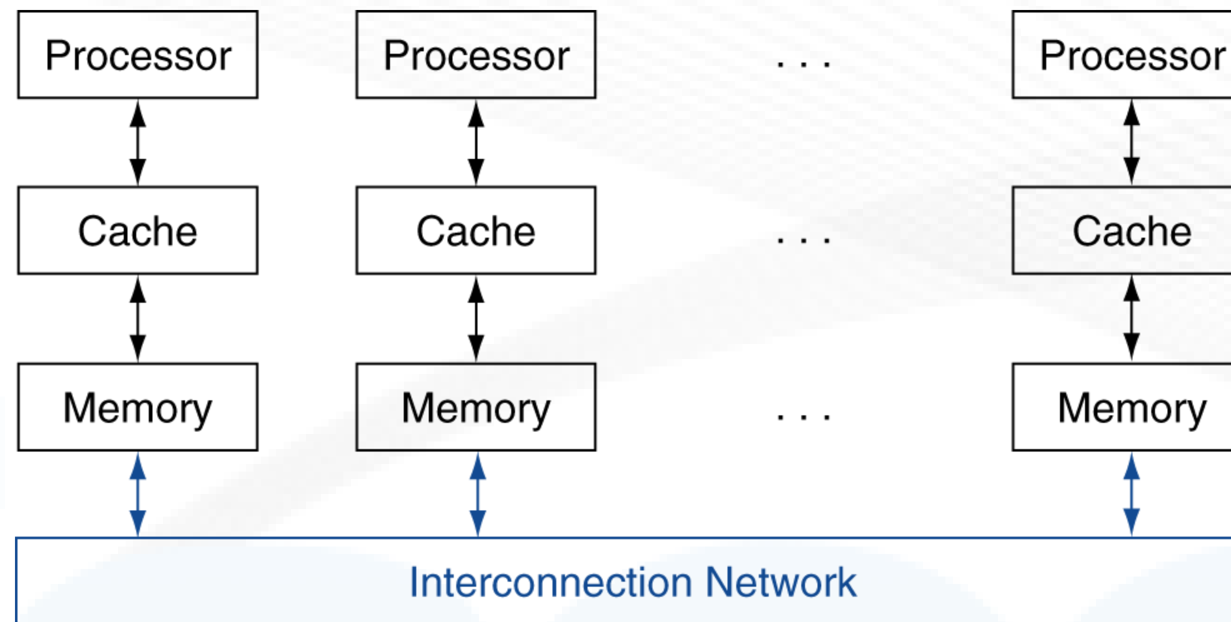


- Transiciones en negro son generadas por el procesador (Pr).
- Transiciones azules son generadas por el Bus (otros procesadores).
- Cada transición muestra el evento que la genera / su resultado.
- Las transiciones al estado M son por escrituras (acierto o fallo).
  - Invalidan el bloque en los demás cachés.
- Cuando un caché tiene un bloque en estado M que es solicitado en el bus, hace un write-back (*flush*).
  - Y pasa el bloque al estado S.
- BusRd vs BusRdX: el primero es por fallos de lectura, el segundo por fallos de escritura (invalida el bloque en los demás caches).

Problema de Falsa coherencia

- Adicionalmente, puede haber otro problema en bloques de múltiples palabras.
- Los movimientos en caché siempre son a nivel de bloques.
- Puede ocurrir que dos variables compartidas no relacionadas X e Y estén ubicadas en el mismo bloque.
  - El procesador A escribe siempre X y el procesador B escribe siempre Y.
  - Ambos están permanentemente invalidando todo el bloque.
  - Generan fallos innecesarios y tráfico por el bus compartido.
- Se puede evitar teniendo mucho cuidado al ubicar las variables en memoria.

Acceso no compartido a memoria



- Las redes de interconexión comunican nodos completos procesador-memoria a través del sistema de I/O.
- Se comunican mediante un mecanismo explícito de **paso de mensajes** (MPI, *Message Passing Interface*).
  - Las aplicaciones empaquetan los datos que quieren transmitir en forma de mensajes.
  - La comunicación es explícita, mediante rutinas para enviar *send*(Pi, Dk) y recibir *wait*(Pj,Dk).
  - Hay librerías que implementan estas rutinas para muchos lenguajes de programación, e iniciativas abiertas como OpenMPI.

Clusters de paso de mensajes

- Las redes de comunicación pueden ser de muy alta performance y muy alto costo.
  - Usadas en supercomputadoras.
- O ser redes de bajo costo (Ethernet).
  - Conectan computadoras completas en configuraciones denominadas **clusters**.
  - Muy populares para ejecutar aplicaciones que aprovechan **paralelismo a nivel aplicación (ALP)**.
    - Búsqueda web, servidor de mail, servidor de archivos, bases de datos.
  - Poseen una alta disponibilidad, a diferencia de SMP.
  - Son fáciles de escalar.

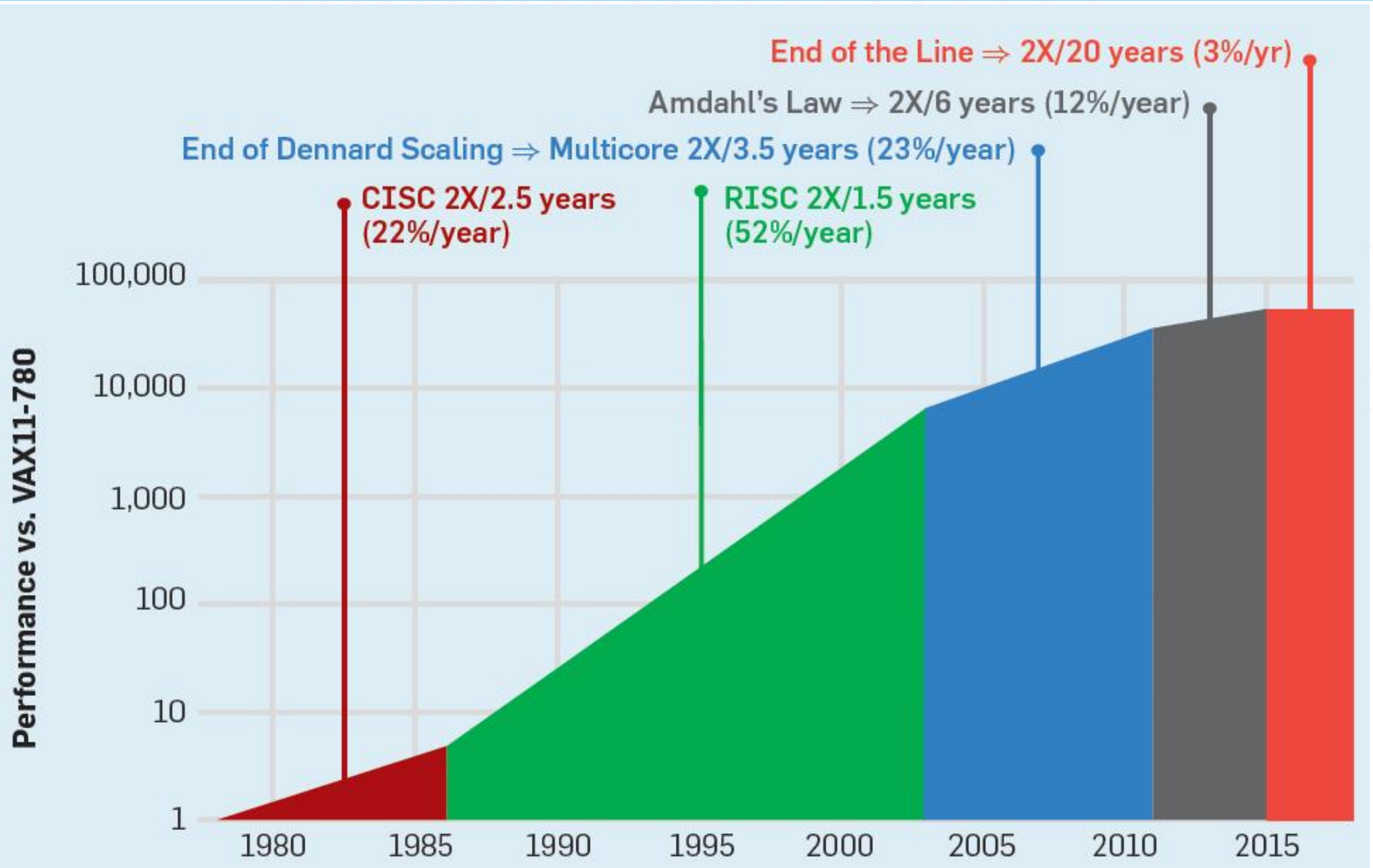
Computadoras de gran escala

- La popularidad de varios servicios relacionados a Internet produjo que algunos clusters sean extremadamente grandes.
  - Reciben el nombre de *Warehouse-Scale Computers* (WSC).
- Su tamaño incrementa la complejidad.
  - Aproximadamente 100.000 servidores.
  - Necesitan espacio físico, energía para alimentarlos y mantenerlos a baja temperatura.
  - Tolerancia a fallos es muy crítica.
- Proveen *Software as a Service* (SaaS).
  - También conocido como la popular “nube” (*cloud computing*).
  - Millones de pequeños pedidos independientes simultáneos que pueden ser paralelizados.
  - Su métrica más importante es la latencia, vista desde el usuario final.

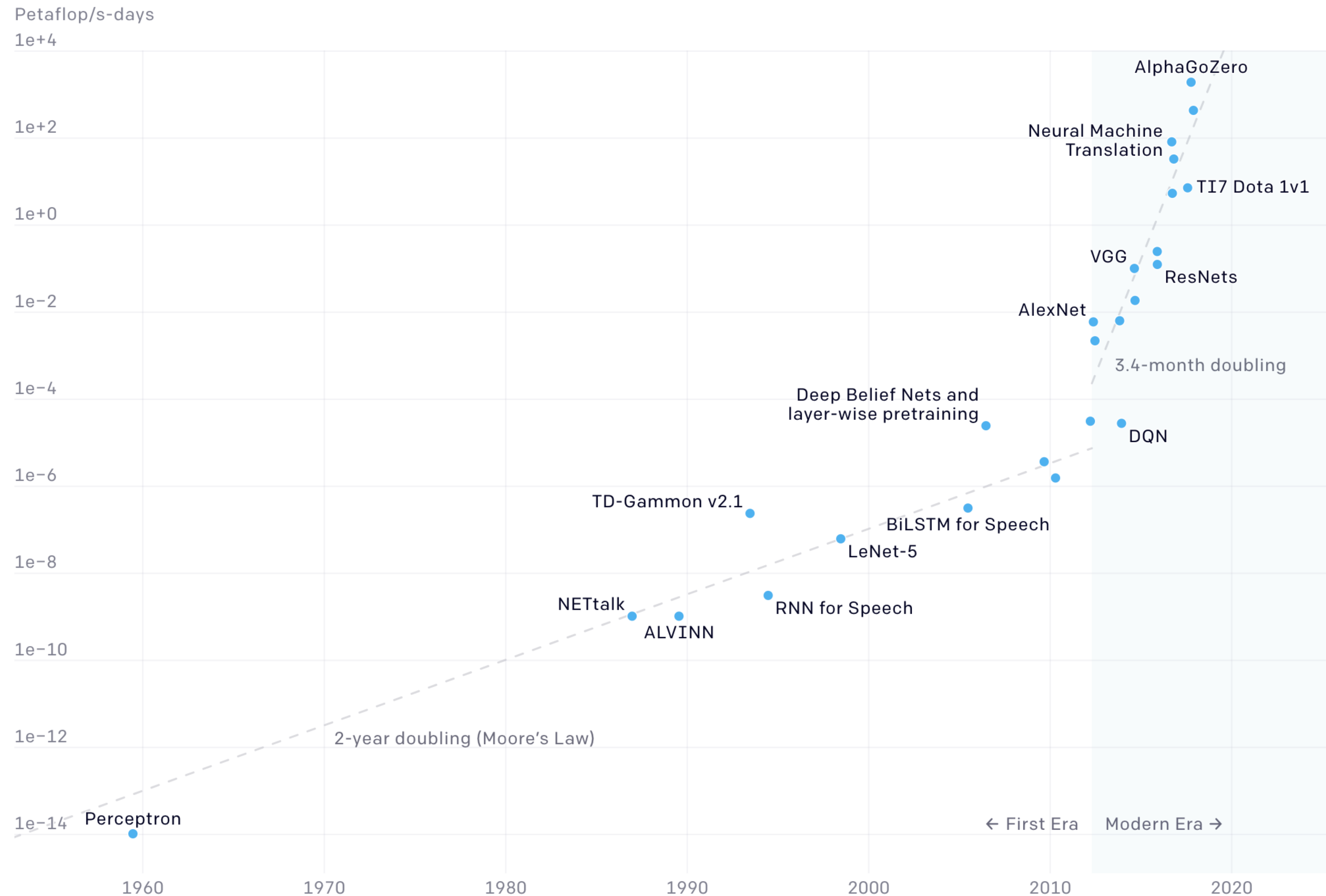
¿Cómo podemos explotar paralelismo?

- Reescribiendo programas con algoritmos/lenguajes que usen paralelismo.
  - Concentrarse en la parte más lenta (Amdahl).
  - Concentrarse en cómo particionar los datos, y cómo hacer la sincronización.
- Distintos enfoques
  - Paralelismo a nivel datos.
  - Paralelismo a nivel thread.
  - Paralelismo a nivel aplicación.
- No escalar paralelismo sin escalar datos.

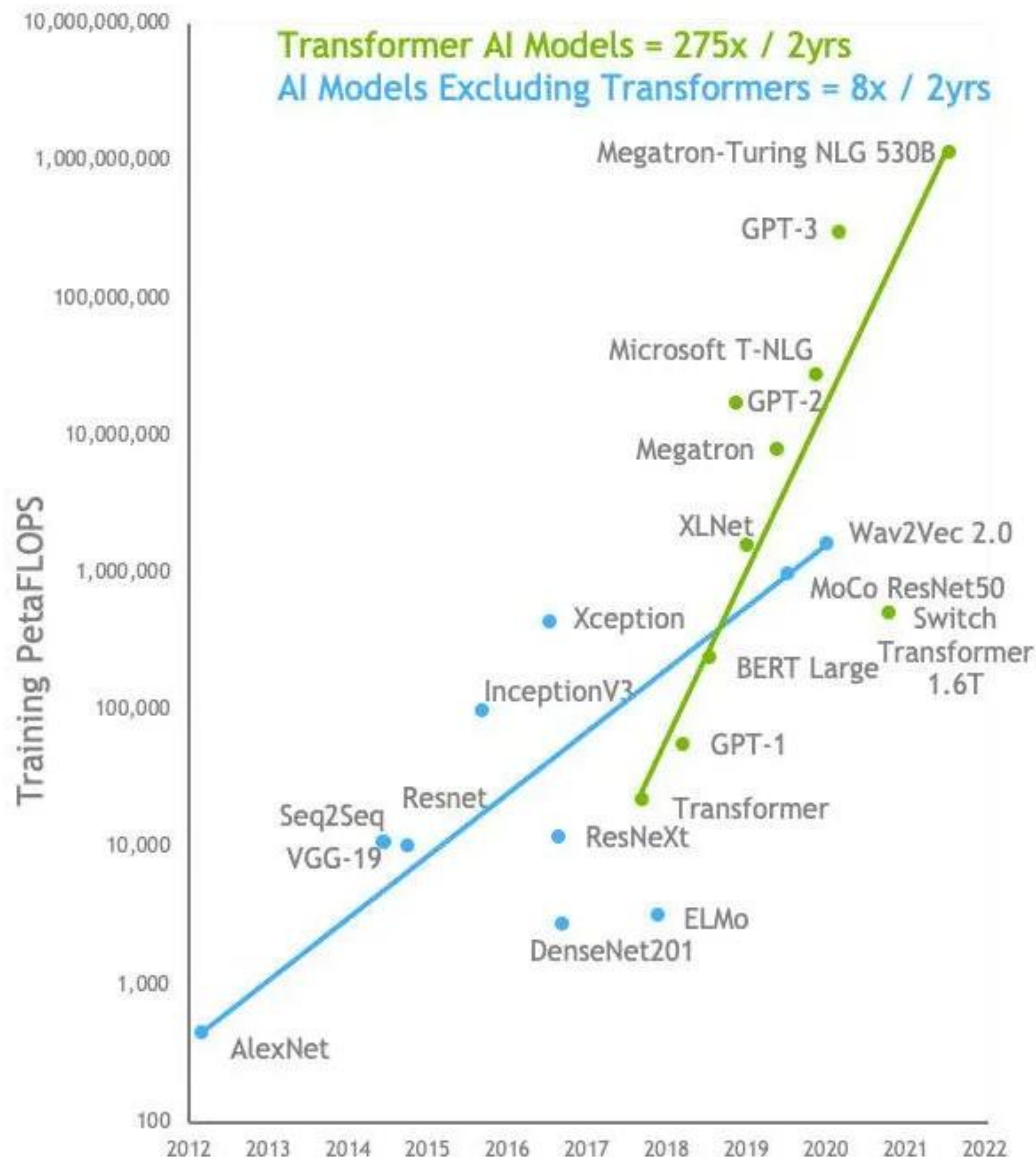
Evolución de la performance



Evolución de necesidades (ej: IA)



Evolución de necesidades (ej: IA)



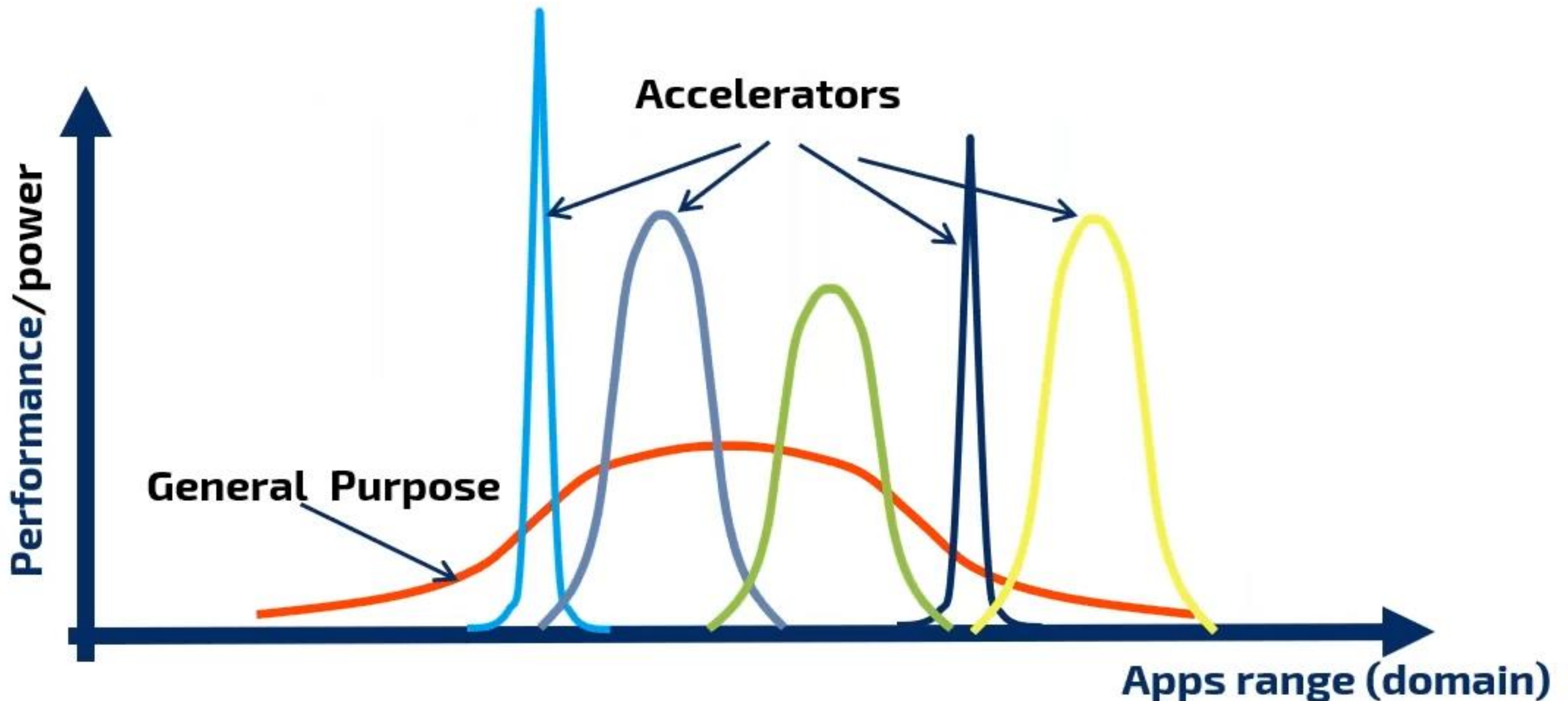
- En la gráfica anterior, performance 2x / 3,5 meses.
  - Eso es 128x / 2 años.
- Según una publicación de NVIDIA (2022), en algunos casos puede ser peor.
  - https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

¿Nuevo cambio de paradigma?

- Según la tendencia mostrada en los gráficos anteriores, los procesadores recién duplicarán su performance en 2038.
  - Pero las necesidades de performance están aumentando más rápido.
  - Las aplicaciones del futuro demandarán mucha más capacidad de cómputo.
- La performance no puede mejorar agregando transistores.
  - Debido al decaimiento de la Ley de Moore, al fin del escalamiento de Dennard y a restricciones de consumo de energía.
- Tampoco mejorará agregando múltiples núcleos.
  - Limitados por la Ley de Amdahl.
- *¿Qué se puede hacer a nivel Arquitectura de computadoras?*
  - La respuesta parece ser **Arquitecturas de Dominio Específico (DSA)**.
  - Hacen muy pocas tareas, pero las hacen extremadamente bien.
  - ¿Recuerdan los IO Processors que vimos en el Tema 13?

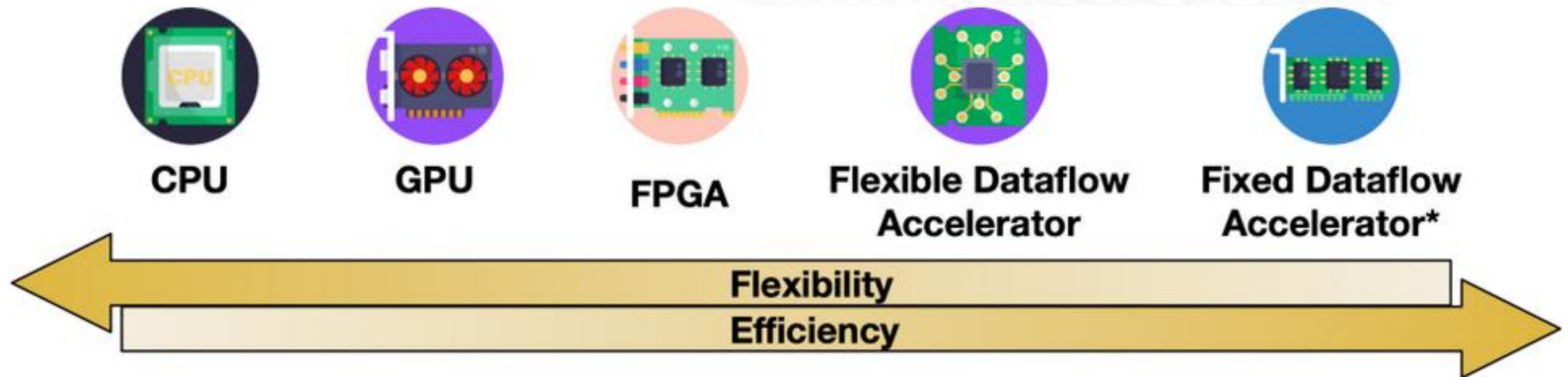
Arquitecturas de Dominio Específico

- Tienen una mejor relación performance/power, pero para una tarea específica.
- Reconfiguran (nuevamente) la Ley de Amdahl.



Arquitecturas de Dominio Específico

- Cambian eficiencia por flexibilidad:



- Cuando están completamente optimizadas para una tarea, si la tarea o sus datos cambian, pierden su ventaja.
  - Lo cual genera un nuevo compromiso: *¿es mejor muy específicos o no tan específicos?*

Arquitecturas de Dominio Específico

- Un ejemplo conocido: CPU vs GPU (del 2018 aprox).

Procesador	Modelo	Max Paralelismo	Frecuencia	Memory Bandwidth	Latencia a Caché L1
CPU	Intel Xeon E5-2690v4	28	3.5 GHz	76.8 GB/s	5 T
GPU	NVIDIA P100	3584	1.1 GHz	732 GB/s	80 T

- GPU con 3x menor frecuencia, pero con 100x paralelismo.
  - Ideal para aplicaciones con mucho paralelismo, más lenta para aplicaciones con poco paralelismo.
- GPU con L1 10x más lento, pero con un ancho de banda a memoria 10x más grande.
  - Ideal para aplicaciones con cálculos predecibles, en los que se pueda anticipar los accesos a memoria.
- Otro ejemplo: un procesador de audio, con un ISA que incluya instrucciones para implementar de manera óptima un algoritmo de cancelación de ruido.

Arquitecturas de Dominio Específico

- Tienen un **paralelismo masivo**: 1000x.
- **Altamente especializadas**:
  - Explotan al máximo el principio de localidad.
  - Optimizan la jerarquía de memorias para estructuras de datos específicas.
  - Reducen todos los *overhead* por comunicaciones al mínimo.
  - Tienen operaciones especiales y tipos de datos especiales. Inclusive pueden tener tamaños de palabra no convencionales.
- Se basan en **un *co-diseño* entre hardware y algoritmos**.
  - Los algoritmos deben ser modificados para aprovechar el nuevo hw.
  - El nuevo hw debe ser diseñado para aprovechar al máximo los algoritmos y sus estructuras de datos.
  - El objetivo es acelerar la solución de un problema visto como un todo.

Arquitecturas de Dominio Específico

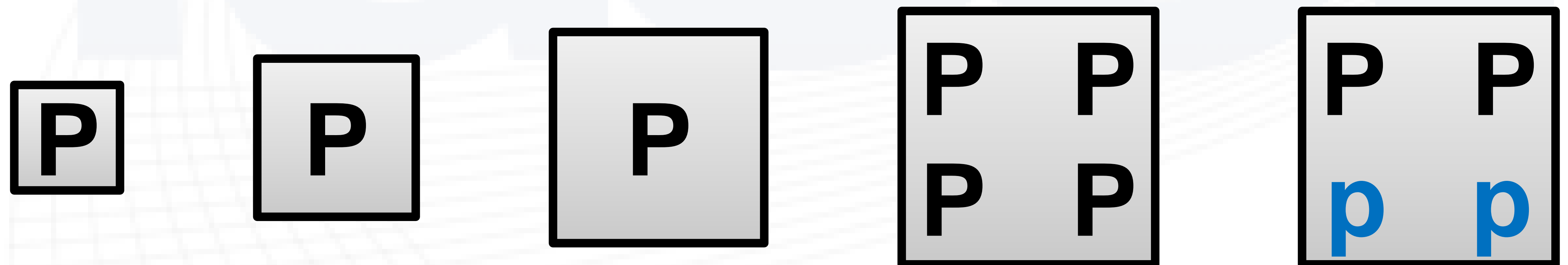
- Desventajas:
  - Dificultad en su programación.
    - Los programadores deben aprender nuevos ISA para cada DSA.
      - *¿Recuerdan al ISA como contrato entre Hw y Sw (Tema 04)?*
    - Las herramientas típicas (compiladores, SO, IDEs, lenguajes, debuggers, etc.) deben ser modificadas para soportar el nuevo hardware.
    - Los programadores deben aprender a usar librerías de paralelismo no convencionales, con abstracciones de alto nivel.
    - Los programadores deben modificar los algoritmos, comprendiendo a fondo qué es lo que hace el hardware.
  - Como son específicas, no disfrutan de las ventajas de la economía de escala, por lo que sus costos fijos las hacen más caras.

Arquitecturas de Dominio Específico

- Ejemplos de problemas donde son mucho mejores:
  - Los cálculos necesarios presentan un alto grado de paralelismo.
  - Los cálculos necesarios presentan regularidad: son muy estables y en intervalos regulares.
  - Los cálculos necesarios presentan mucha localidad, necesitando pocos accesos a memoria.
  - Los cálculos necesarios pueden hacerse con menos dígitos de precisión.
- Ejemplos donde no son mejores:
  - Aplicaciones donde los algoritmos actuales no admiten paralelismo (Bases de Datos transaccionales).
  - Aplicaciones con demanda insuficiente: investigaciones científicas, como estudios climáticos o genéticos.

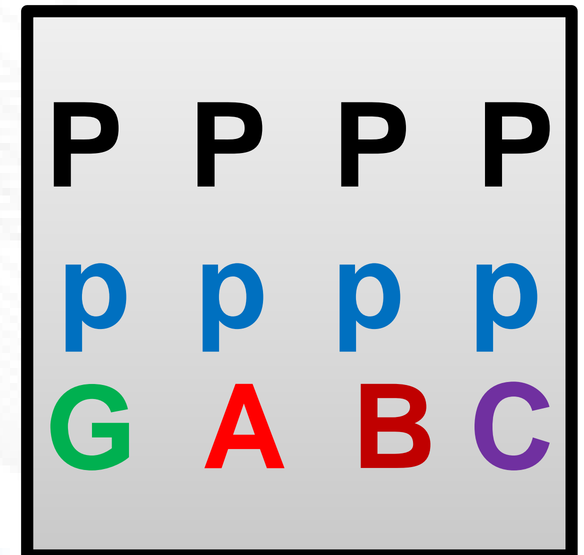
¿Nuevo cambio de paradigma?

- Hasta hoy en día, los procesadores (y los sistemas) se diseñan bajo la guía de “las tres P”.
  - Pero hay otro factor implícito: **generalidad**. Deben poder ejecutar (ser compatibles) cualquier aplicación que el usuario final requiera.
- Antes, un procesador nuevo anunciaba cuántos transistores traía.
- Luego, se anunciaba cuántos núcleos homogéneos traía.
- Luego, los núcleos ya no fueron homogéneos.



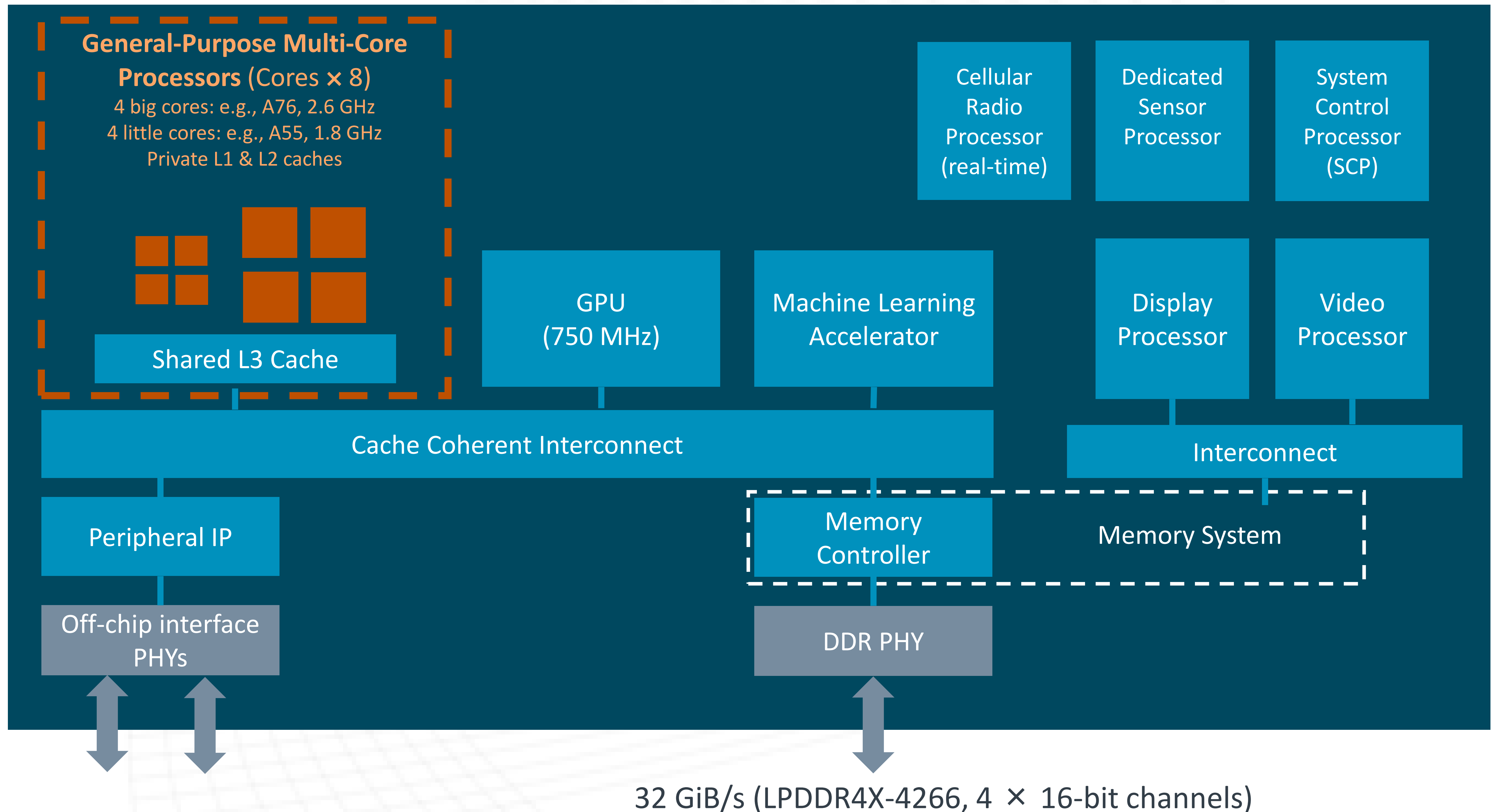
¿Nuevo cambio de paradigma?

- Actualmente, es una **mezcla heterogénea de varios núcleos**.
  - Cada núcleo dedicado a una aplicación, con su propio ISA, su propio software, sus propios algoritmos.
  - ¿Unificar los ISA para disminuir la fragmentación? **RISC-V**.



- Estas Arquitecturas de Dominio Específico están siendo dominantes o están en pleno crecimiento
  - En todos los tipos de computadoras (desktop, móviles, servidores, datacenters y sistemas embebidos).

Ejemplo de un SoC con varios DSA



Ejemplos de DSA

- El Procesador M1 de Apple (2020) contiene:
  - 4 procesadores de alta performance (Firestorm) + 4 procesadores de alta eficiencia (Icestorm).
  - 1 procesador de video (GPU) de 8 núcleos + 1 procesador para redes neuronales de 16 núcleos.
  - 1 procesador para procesamiento de imágenes + 1 procesador para seguridad.
- Pasando a Google:
  - Diseñó sus propios chips para IA en 2016 (TPU). En 2021 anunció la 4ta versión.
  - En 2021 anunció que diseñó sus propios chips para video (VPU), específicamente para códecs de YouTube.
- Otro ejemplo: ¿se acuerdan del procesador *WaferScale* que vimos en el Tema 02?
  - El chip más grande de la historia ejecuta una única tarea.
  - Un sistema de 32 chips posee 27 millones de núcleos, y puede procesar hasta 2 exaFLOPS con un consumo de solamente 16 kW.

Repaso: ¿Para qué aprender esos temas?

Apple Releases M2 Pro and M2 Max: 20 Percent Faster, Up to 19 GPU Cores

By Brandon Hill published January 17, 2023

(Image credit: Apple)

- Comprender a fondo la noticia.
- Discernir cuán cierta es.
- Discernir cuán útil me resulta.
- Discernir qué costo estaría dispuesto a pagar.

Resumen final

- Multiprocesamiento como técnica para seguir aumentando la performance.
  - Superar limitaciones de ILP, *Memory Wall* y *Power Wall*.
- Problemas: comunicación, sincronización, coherencia, dificultad.
- Performance limitada por software (Amdahl).
- No tiene sentido aumentar paralelismo sin escalar los datos.
- SIMD para aprovechar paralelismo a nivel datos.
  - Multimedia, grandes volúmenes de datos.
- Multithreading simultáneo para aprovechar paralelismo a nivel threads.

Resumen final

- SMP de acceso a memoria compartido y uniforme.
  - Genera problemas de coherencia de cachés.
  - Es necesario un protocolo implementado mediante una MEF.
- Existen sistemas de acceso a memoria compartida no uniforme (NUMA).
- Clusters de acceso no compartido a memoria, que se comunican mediante paso de mensajes.
  - Interconexión de computadoras completas, mediante redes de bajo costo, con alta disponibilidad.
  - Escalaron a WSC que proveen Cloud Computing.
- Reciente estancamiento en la mejora de performance.
  - Posible nuevo cambio de paradigma.
  - Arquitecturas de Dominio Específico (DSA).

Agradecimientos

- Las diapositivas de este tema fueron basadas en las realizadas por el Ing. Daniel Cohen.
- A su vez inspiradas en las diapositivas tomadas de:
  - Curso CS152, Berkeley, 2006, John Lazzaro.
  - Conferencia de David Patterson, Berkeley, 2006.
  - Curso CS152, Berkeley, 2008, Krste Asanovic
  - Curso EE252, Stanford University, Christos Kozirakys.