A Journey from Cache Coherence to Microarchitecture

Alberto Ros

University of Murcia, Spain

Jun 17, 2026

MY FIRST (NO) PAPER

XV JORNADAS DE PARALELISMO—ALMERIMAR, SEPTIEMBRE 2004

Mejorando la escalabilidad en arquitecturas cc-NUMA: El directorio *ligero*

Alberto Ros, Manuel E. Acacio y José M. García

Resumen— **Uno de los problemas más importantes en los multiprocesadores escalables de memoria compartida es la cantidad de memoria necesario para almacenar la información de directorio, dicha cantidad puede llegar a mantener la coherencia de las caches. Es deseable que la sobrecarga que introduce esta memoria sobre el total de la máquina sea lo más pequeña posible y además que escale más lentamente conforme aumente el número de procesadores. Desafortunadamente, las arquitecturas que reducen el tamaño de esta memoria empeoran el rendimiento. Este artículo presenta una nueva arquitectura que reduce considerablemente el tamaño necesario para la información de directorio y, además, obtiene una mejora en el rendimiento. A esta arquitectura la hemos llamado directorio ligero y elimina completamente la estructura de directorio de la memoria principal. Ahora la información de directorio estará sólo a nivel de la cache, que posee muchas menos líneas que la memoria principal, con lo que se logra una reducción en el tamaño considerable, además de un acceso más rápido a la información de directorio.**

Palabras clave— **Directorio Ligero, Escalabilidad, Mejora de prestaciones, Multiprocesadores cc-NUMA.**

I. Introducción y motivación

En la actualidad, la mejor opción para diseñar multiprocesadores escalables son las llamadas arquitecturas cc-NUMA, basadas en una red de interconexión punto a punto [1] y la memoria principal físicamente distribuida para garantizar que el acceso a la memoria sea escalable conforme aumenta el número de procesadores. Sin embargo, estos multiprocesadores tienen dos problemas para lograr escalabilidad y rendimiento.

Al ser su red de conexión punto a punto, un nodo no puede observar la información que se dirige a otro nodo, y por tanto, para mantener la coherencia de la información en las caches necesita una estructura auxiliar llamada directorio [2], que contiene la información de qué líneas de memoria están en cache y qué nodos poseen dichas líneas.

Una forma natural (de aquí en adelante básica) de organizar este directorio consiste en tener una entrada de directorio para cada línea de la memoria principal, y por cada entrada dos campos. Uno de ellos indicará el estado del dato, y será de tamaño pequeño (2 bits normalmente), y el otro, llamado código de compartición, que indica qué nodos comparten ese dato. La forma más sencilla de almacenar el código de compartición se conoce como *full-map* o *bit-vector* [2], y consta de un bit para cada nodo que indica si dicho nodo está compartiendo el dato. El problema consiste en que este campo aumenta linealmente con el número de nodos de la máquina y por tanto cada vez se introducirá más sobrecarga en esta memoria, impidiendo la escalabilidad de la máquina. El segundo problema consiste en que al estar el directorio en la memoria principal, el tiempo de acceso a dicha estructura es bastante elevado y, teniendo en cuenta que siempre que se recibe una petición hay que acceder a memoria para ver el estado de la línea, se empeora el rendimiento de los protocolos.

En este estudio se pretende diseñar y evaluar una nueva clase de directorio basada en eliminar de la memoria principal, con el fin de reducir la sobrecarga de memoria y lograr mejorar la escalabilidad de este tipo de multiprocesadores.

La mayor parte de la memoria de directorio de los multiprocesadores actuales no se usa. Esto es debido a que el directorio sólo contiene información útil cuando los datos están en alguna cache. Los datos no cacheados tienen asignada una porción de memoria que no se está usando en ese instante. Por tanto, si almacenáramos sólo la información de directorio en las respectivas líneas de cache, podríamos prescindir de esta información en la memoria principal.

A esta idea la hemos denominado *directorio ligero*, y se basa en situar la estructura del directorio en la cache de segundo nivel, mucho más pequeña que la memoria principal y con menor tiempo de acceso. Con esto se logra reducir la cantidad de memoria necesaria para mantener la coherencia y, por tanto, mejorar la escalabilidad de los multiprocesadores cc-NUMA. Ahora los datos que no estén en ninguna cache no estarán ocupando memoria en el directorio.

Para analizar la viabilidad de esta idea hemos realizado unas simulaciones sobre un conjunto de aplicaciones por medio de RSIM [3]. Dicho conjunto está formado por `EM3D`, `FFT`, `MP3D`, `OCEAN`, `UNSTRUCTURED` y `WATER-NSQ`. Para todas ellas se han utilizado 32 nodos y se han tomado únicamente datos referentes a esa parte paralela.

Una consecuencia de la eliminación del directorio de la memoria principal es una mejora en la eficiencia del multiprocesador.

En primer lugar, el hecho de tener la información de directorio en la cache nos reduce la latencia de acceso a la información. Esto repercute cuando la línea ya ha sido accedida por el nodo al que pertenece la línea de memoria (*home*) y por tanto ya se ha traído la línea a su cache. En este caso ya no tendríamos que ir a buscar la información de directorio a la memoria principal. El número de veces que se produce este hecho para el caso de un directorio convencional, se muestra en la figura 1 (barra etiquetada con básico).

En el caso del protocolo ligero, al tener en la cache las líneas con la información de directorio que hayan sido necesitadas por algún nodo, aumentará el número de veces que, al llegar una petición al nodo *home*, se encontrará la línea en la cache, con lo que nos ahorraremos nuevamente el tener que acceder a la memoria principal a buscar la información de directorio. En la figura 1 (barra etiquetada con ligero) se observa como la mayoría de las veces que se necesita una línea ya está en la cache del nodo *home*, teniendo en cuenta el protocolo de directorio ligero. En realidad el número de veces que no estaría corresponde a reemplazos en el *home*, ya que no se tienen en cuenta los fallos en frío que se producen en la parte de inicialización de la aplicación.

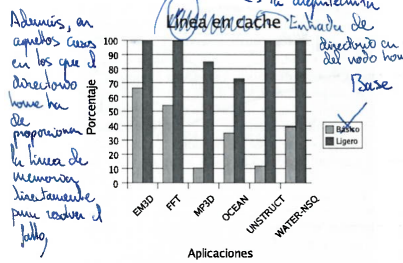


Fig. 1. Estimación de las veces que la información de directorio estará en la cache en un protocolo de directorio básico y ligero

Además si la petición es una lectura de un dato, éste estará disponible en la cache del nodo *home* para las peticiones posteriores. En la figura 2 se muestra el porcentaje de veces que la cache del nodo *home* podría proporcionar el dato que se le ha solicitado, teniendo en cuenta que la información de directorio está almacenada en dicha cache (arquitectura de directorio ligero propuesta).

Pero también existe una parte negativa, y consiste en que cada vez que llega a la cache de su nodo *home* por un dato pedido por otro nodo, podría desplazar un dato de la cache que probablemente esté siendo utilizado. Para evaluar como afecta de perjudicial este efecto hemos medido el número de líneas que llegan a cada cache para el protocolo básico y las que llegan para el ligero. En la tabla I se muestra la sobrecarga del tráfico sobrecargado. Se puede apreciar que esta sobrecarga es mínima, con lo que esta parte negativa del protocolo pierde importancia. La aplicación `OCEAN` es la que contiene el nodo con sobrecarga, un 19%. El resto de las aplicaciones prácticamente no producen sobrecarga de líneas al aplicar el nuevo protocolo.

La diferencia entre el número de líneas del protocolo de directorio básico y ligero consiste en el hecho de llevar la cache del nodo *home* líneas de memoria que han sido accedidas por otros nodos pero

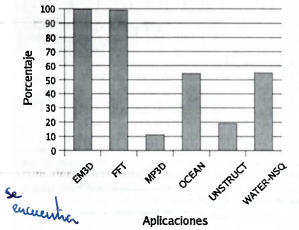


Fig. 2. Estimación de las veces que la cache del *home* puede proporcionar el dato pedido

TABLA I
SOBRECARGA DE LA CACHE DEBIDA AL PROTOCOLO LIGERO

Aplicación	Básico	Ligero	Sobrecarga
EM3D	814	814	1.0000
FFT	2355	2356	1.0004
MP3D	1018	1046	1.0275
OCEAN	15196	18088	1.1902
UNSTRUCT	1587	1593	1.0038
WATER-NSQ	1399	1421	1.0157

número de líneas que han sido accedidas por un nodo y no han sido utilizadas en ningún momento por su nodo *home*, es decir las líneas innecesarias que se habrían almacenado en las caches L2. Este estudio no considera los reemplazos, por lo que estos números se darían en el caso de una cache L2 infinita y se usan únicamente para obtener una ligera información de como puede resultar de beneficioso el protocolo.

Finalmente, con el fin de obtener una estimación del *SpeedUp* que se obtendría con la arquitectura propuesta se ha realizado una estimación utilizando el tiempo de ejecución de las aplicaciones para el directorio ligero y el básico. La tabla II muestra la mejora obtenida para el caso base.

TABLA II
ESTIMACIÓN DEL SPEEDUP QUE SE PODRÍA OBTENER CON LA ARQUITECTURA DE DIRECTORIO LIGERO

Aplicación	Básico	Ligero	SpeedUp
EM3D	26329229	25391169	1.04
FFT	6589370	6255173	1.05
MP3D	19576568	9794022	2.00
OCEAN	74096404	60826158	1.22
UNSTRUCT	207440536	183172663	1.13
WATER-NSQ	35389594	33901000	1.04

Por todos los resultados mostrados, la arquitectura de directorio ligero permitiría eliminar el directorio de la memoria principal, haciéndolo escalable, además obtendría un aumento en el rendimiento.

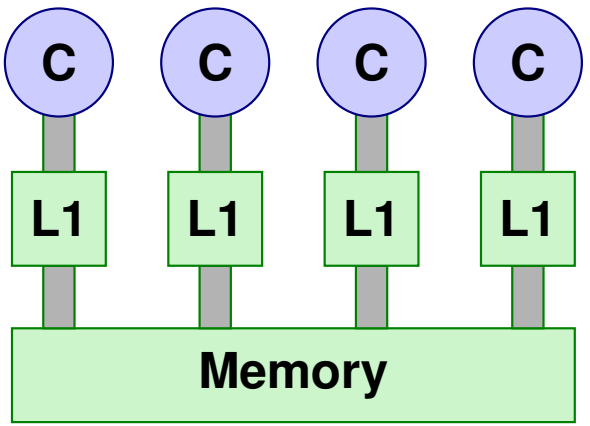
Por tanto podría ser una alternativa viable para los futuros multiprocesadores

Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, Campus Universitario de Espinardo, Murcia (España). Correos electrónicos: a.ros@ditec.um.es, meacacio@ditec.um.es, jmgarcia@ditec.um.es

CACHE COHERENCE PROTOCOLS

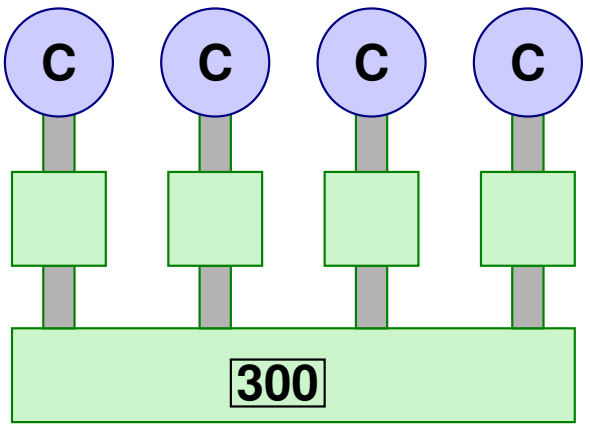
CACHE COHERENCE IN A NUTSHELL

- Caches are fundamental for high performance
- In multicore processors caches can cause incoherences
  - → Two cores see different values of a data at the same time
- Example: bank account



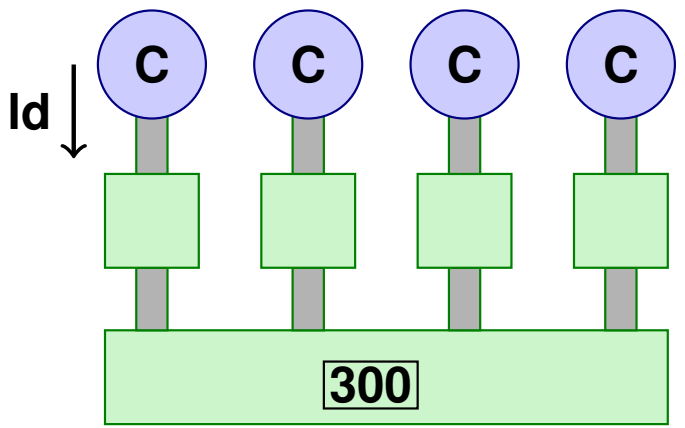
CACHE COHERENCE IN A NUTSHELL

- Caches are fundamental for high performance
- In multicore processors caches can cause incoherences
  - → Two cores see different values of a data at the same time
- Example: bank account



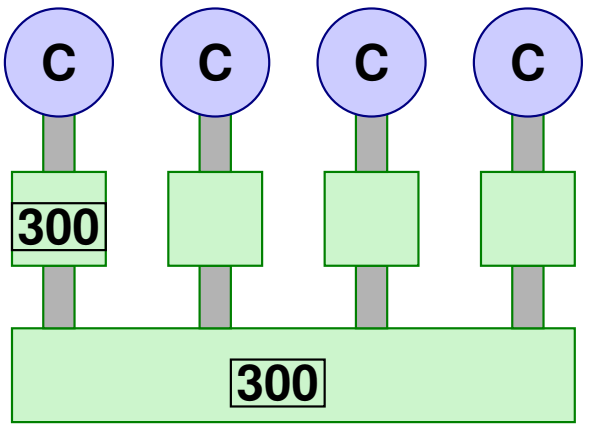
CACHE COHERENCE IN A NUTSHELL

- Caches are fundamental for high performance
- In multicore processors caches can cause incoherences
  - → Two cores see different values of a data at the same time
- Example: bank account



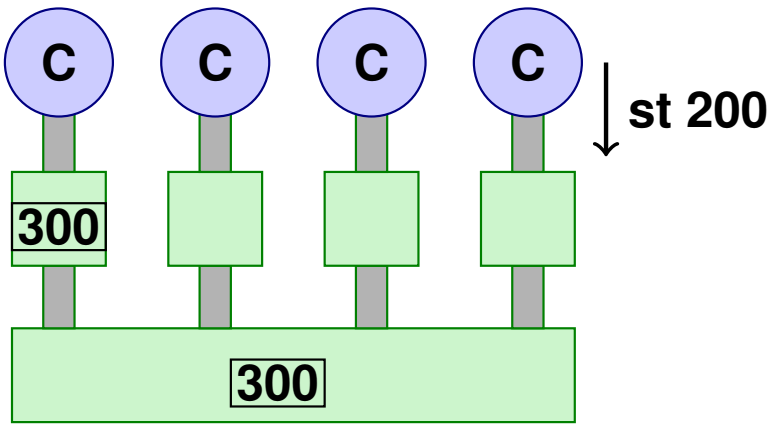
CACHE COHERENCE IN A NUTSHELL

- Caches are fundamental for high performance
- In multicore processors caches can cause incoherences
  - → Two cores see different values of a data at the same time
- Example: bank account



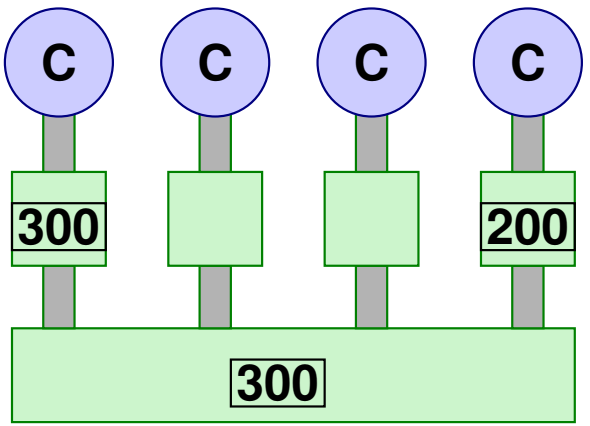
CACHE COHERENCE IN A NUTSHELL

- Caches are fundamental for high performance
- In multicore processors caches can cause incoherences
  - → Two cores see different values of a data at the same time
- Example: bank account



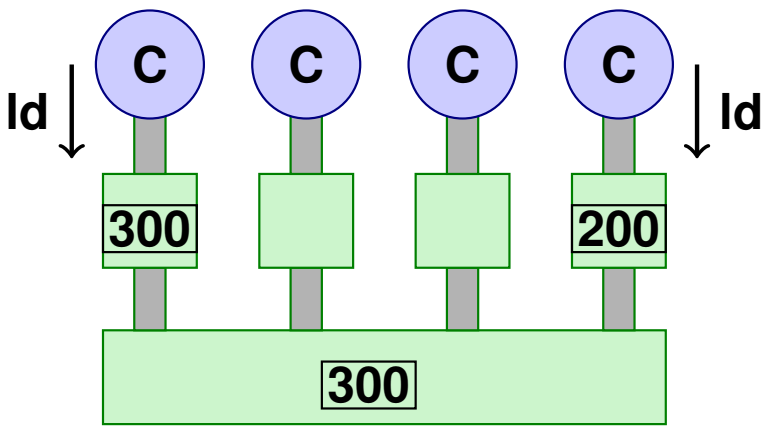
CACHE COHERENCE IN A NUTSHELL

- Caches are fundamental for high performance
- In multicore processors caches can cause incoherences
  - → Two cores see different values of a data at the same time
- Example: bank account



CACHE COHERENCE IN A NUTSHELL

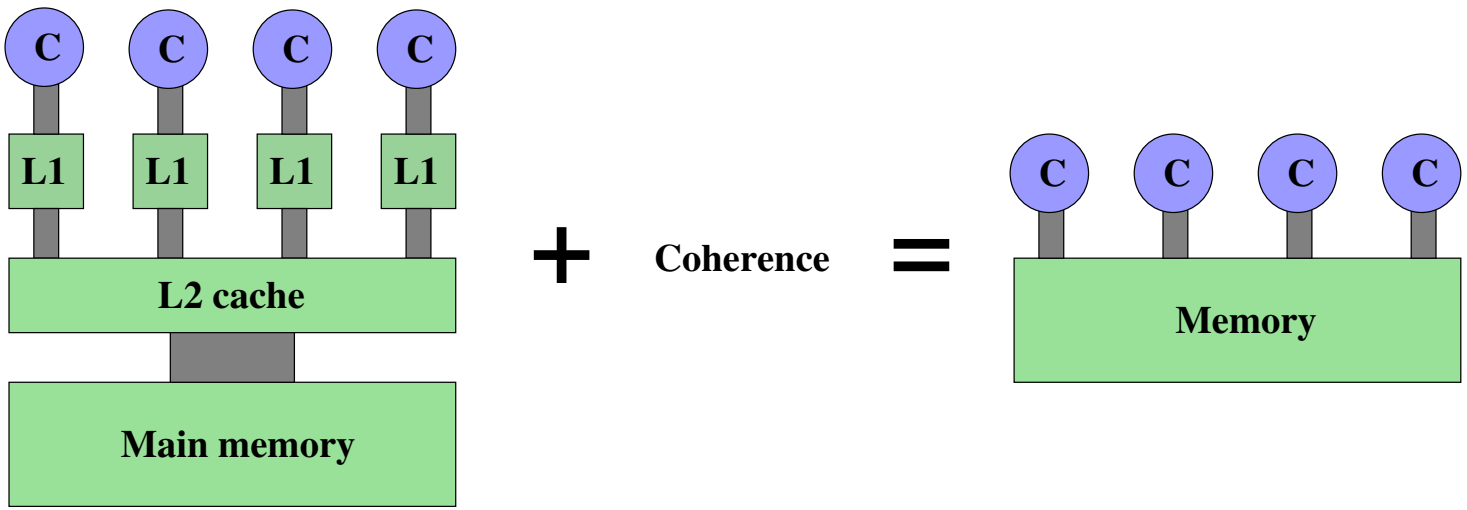
- Caches are fundamental for high performance
- In multicore processors caches can cause incoherences
  - → Two cores see different values of a data at the same time
- Example: bank account



- Incoherence!

CACHE COHERENCE IN A NUTSHELL

- The cache coherence protocol makes caches in a shared memory functionally invisible



- Data appear to be in a single location
- Operates per address $\rightarrow$ each address is treated as independent

COHERENCE INVARIANTS

Two invariants suffice to accomplish the coherence goal[1]

[1] Dubois et al. “Parallel Computer Organization and Design”, 2012

COHERENCE INVARIANTS

Two invariants suffice to accomplish the coherence goal[1]

1. SWMR: SINGLE-WRITER–MULTIPLE-READER

For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it

[1] Dubois et al. “Parallel Computer Organization and Design”, 2012

COHERENCE INVARIANTS

Two invariants suffice to accomplish the coherence goal[1]

1. SWMR: SINGLE-WRITER–MULTIPLE-READER

For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it



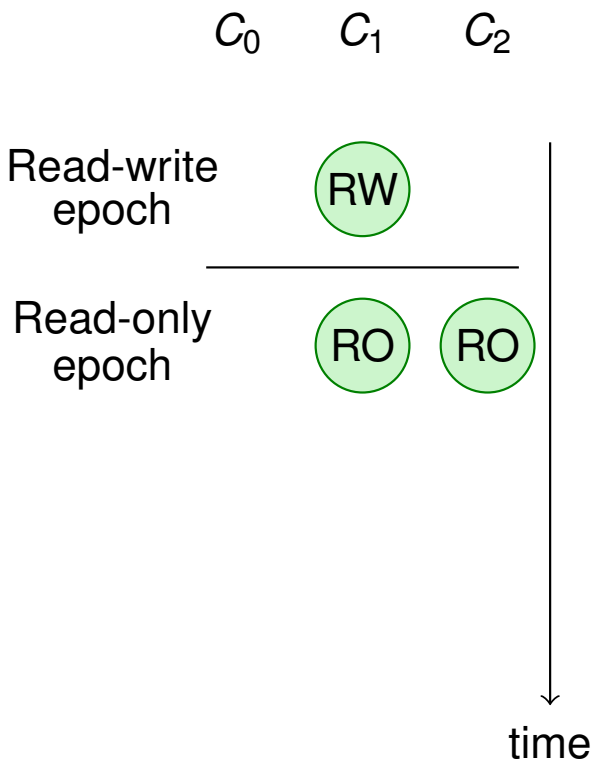
[1] Dubois et al. “Parallel Computer Organization and Design”, 2012

COHERENCE INVARIANTS

Two invariants suffice to accomplish the coherence goal[1]

1. SWMR: SINGLE-WRITER–MULTIPLE-READER

For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it



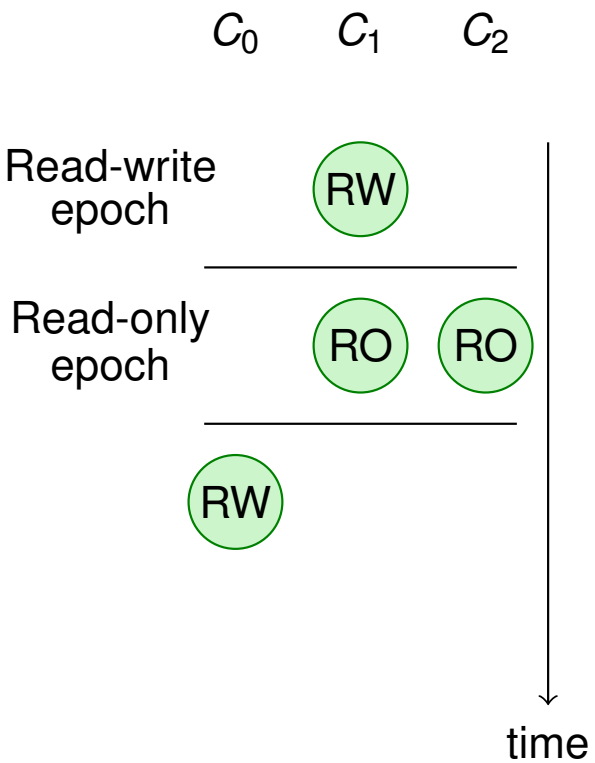
[1] Dubois et al. “Parallel Computer Organization and Design”, 2012

COHERENCE INVARIANTS

Two invariants suffice to accomplish the coherence goal[1]

1. SWMR: SINGLE-WRITER–MULTIPLE-READER

For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it



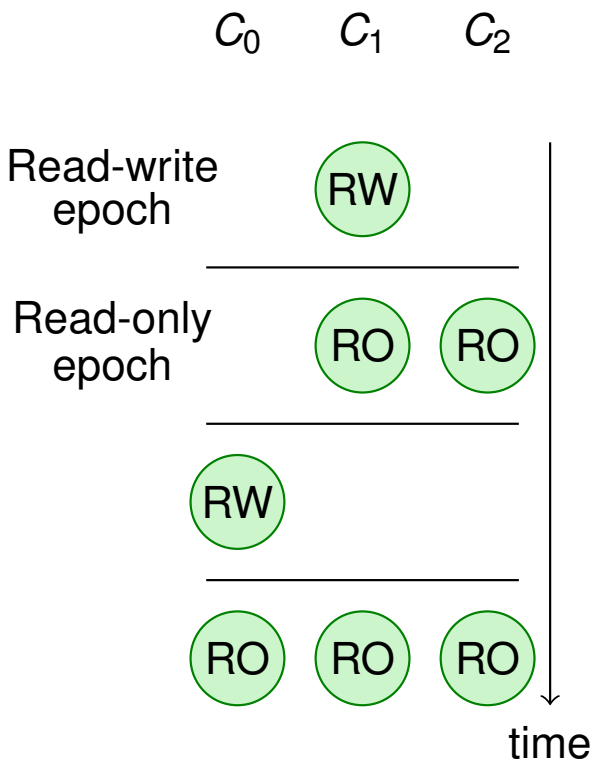
[1] Dubois et al. “Parallel Computer Organization and Design”, 2012

COHERENCE INVARIANTS

Two invariants suffice to accomplish the coherence goal[1]

1. SWMR: SINGLE-WRITER–MULTIPLE-READER

For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it



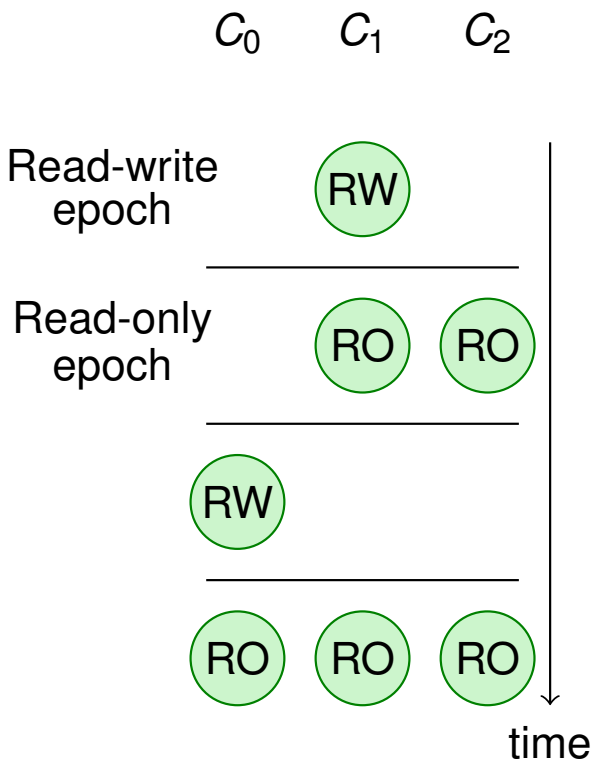
[1] Dubois et al. “Parallel Computer Organization and Design”, 2012

COHERENCE INVARIANTS

Two invariants suffice to accomplish the coherence goal[1]

1. SWMR: SINGLE-WRITER–MULTIPLE-READER

For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it



2. DATA VALUE

The value of a memory location at the start of an epoch is the same as the value of the memory location at the end of its last read–write epoch

[1] Dubois et al. “Parallel Computer Organization and Design”, 2012

COHERENCE INVARIANTS

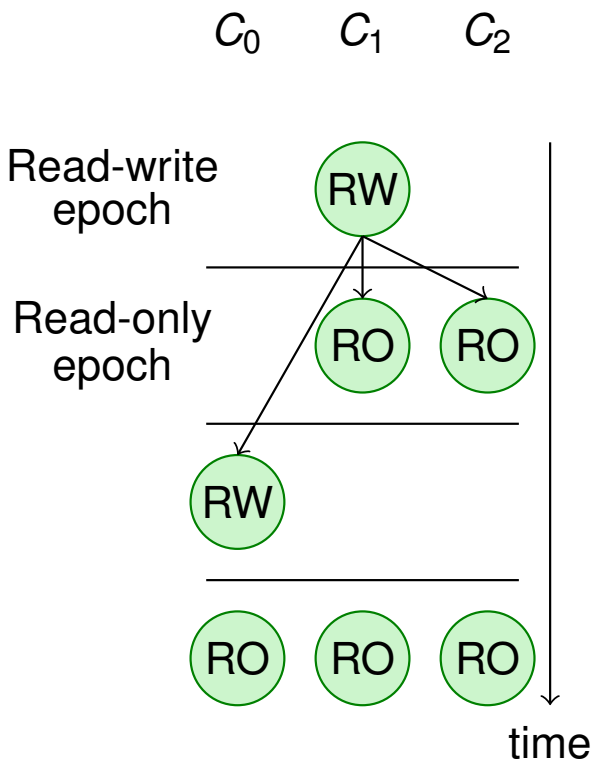
Two invariants suffice to accomplish the coherence goal[1]

1. SWMR: SINGLE-WRITER–MULTIPLE-READER

For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it

2. DATA VALUE

The value of a memory location at the start of an epoch is the same as the value of the memory location at the end of its last read–write epoch



[1] Dubois et al. “Parallel Computer Organization and Design”, 2012

KEY PROBLEMS IN CACHE COHERENCE IMPLEMENTATIONS

- SWMR INVARIANT ussually achieved through data invalidation
- DATA-VALUE INVARIANT achieved through cache-to-cache forwarding
  - → Cost in traffic and latency (indirection)

- Efficient invalidation require sharers tracking: the directory
  - → Cost in storage and not scalable

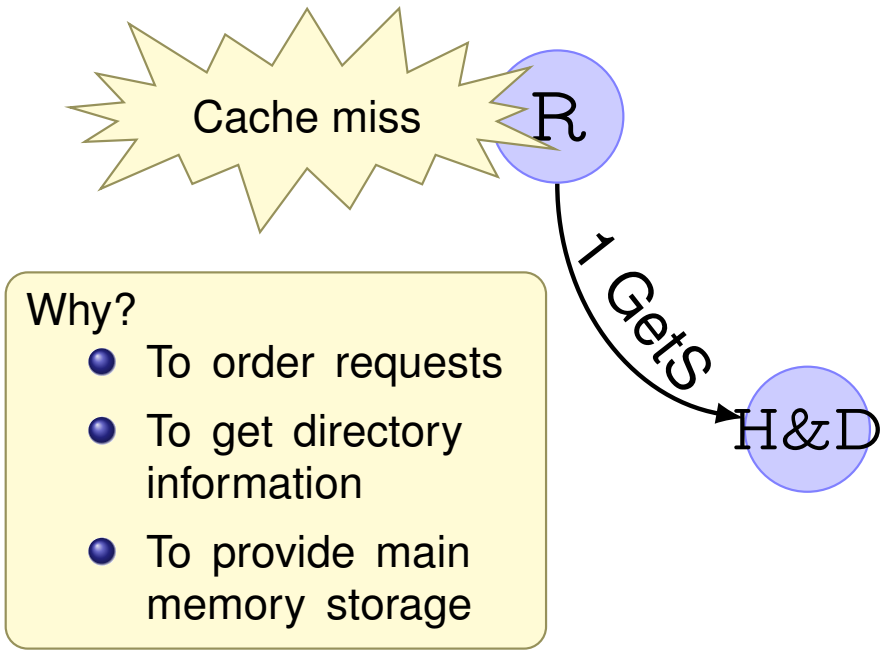
INDIRECTION PROBLEM: LATENCY/TRAFFIC

CACHE-TO-CACHE TRANSFER IN DIRECTORY-BASED PROTOCOLS



INDIRECTION PROBLEM: LATENCY/TRAFFIC

CACHE-TO-CACHE TRANSFER IN DIRECTORY-BASED PROTOCOLS

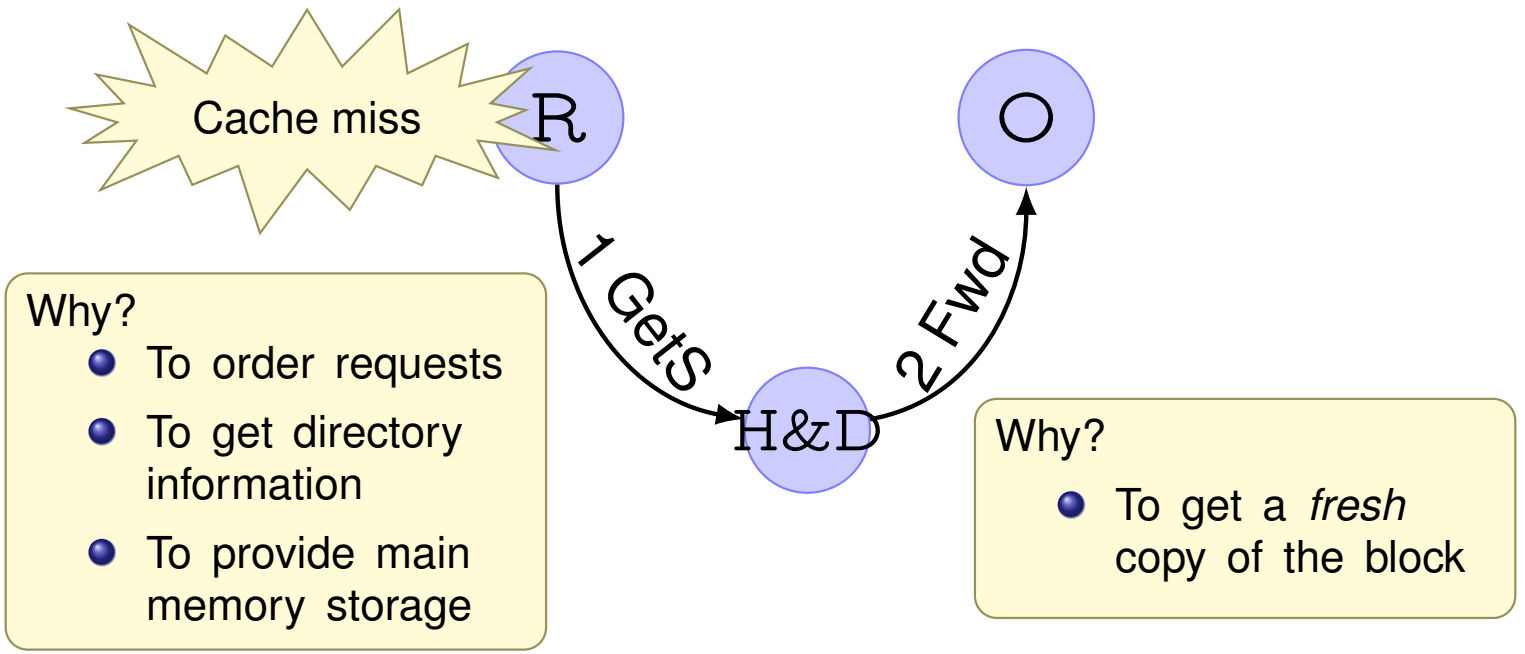


Why?

- To order requests
- To get directory information
- To provide main memory storage

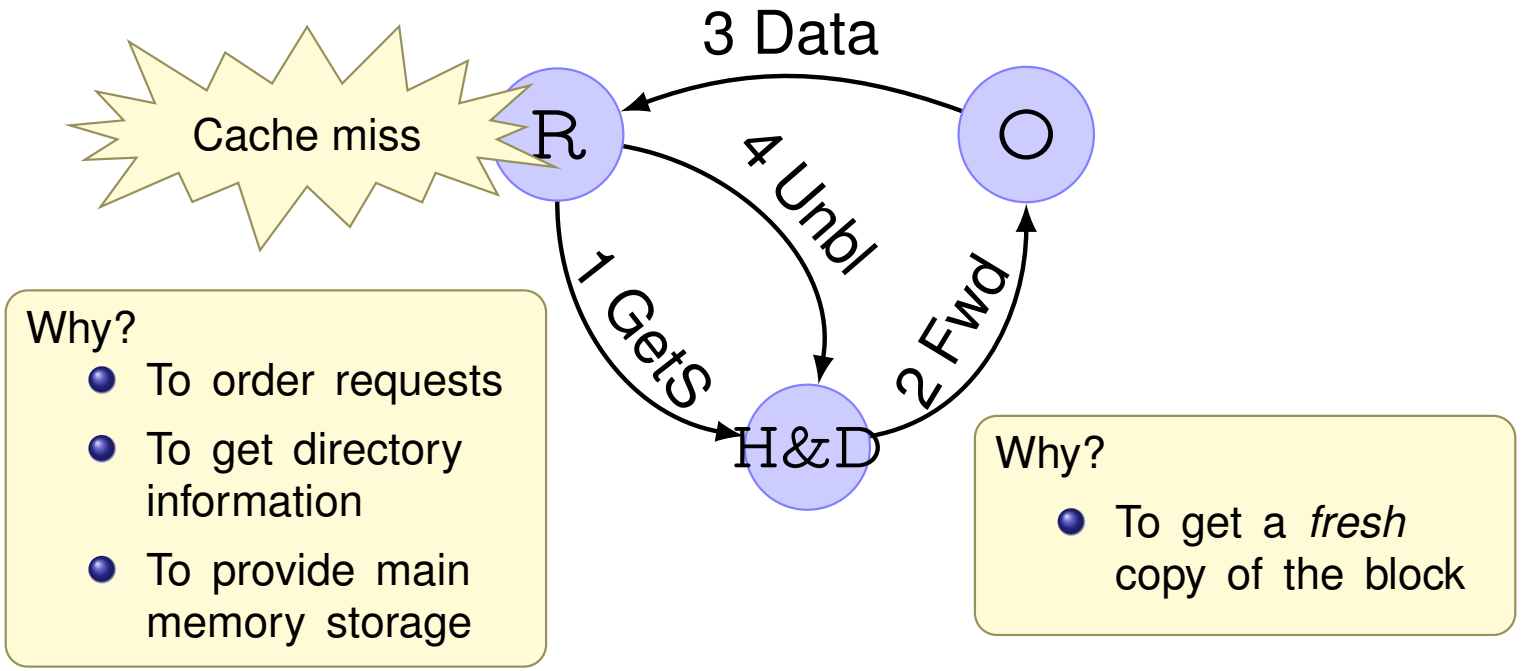
INDIRECTION PROBLEM: LATENCY/TRAFFIC

CACHE-TO-CACHE TRANSFER IN DIRECTORY-BASED PROTOCOLS



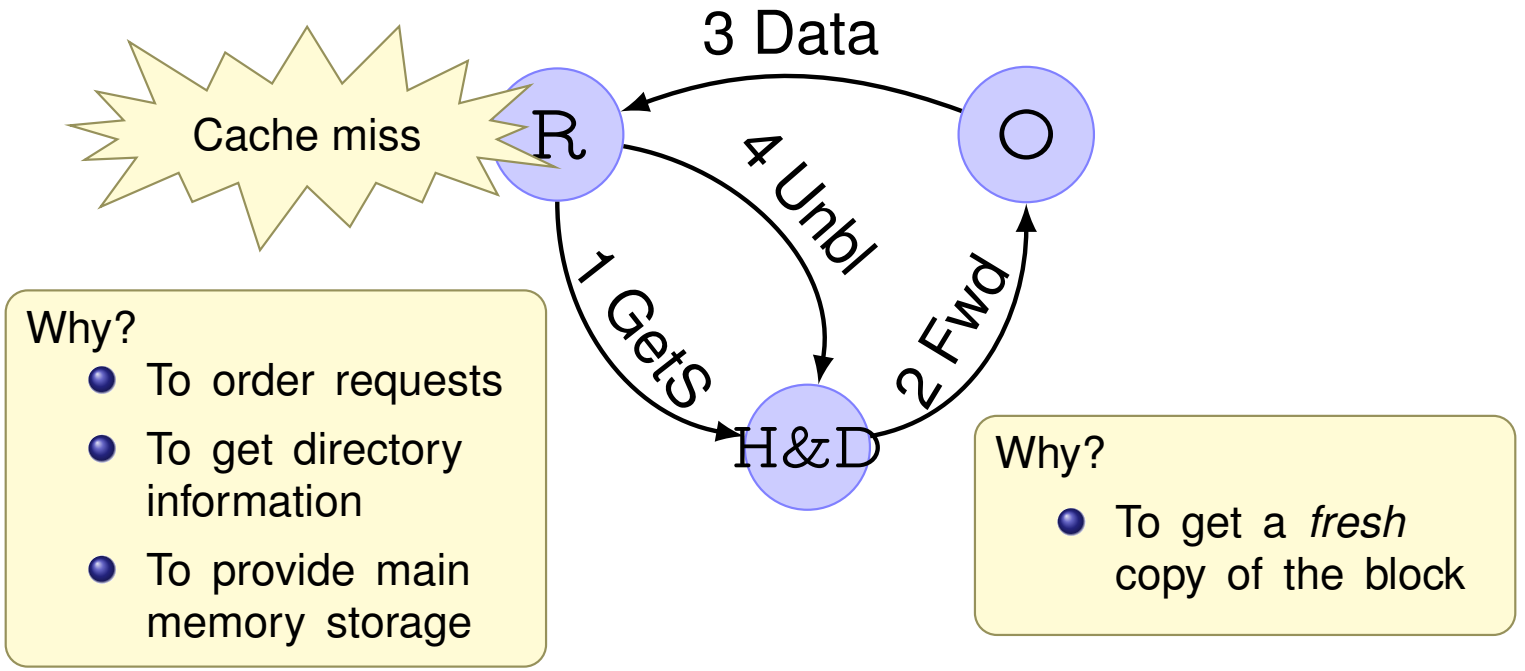
INDIRECTION PROBLEM: LATENCY/TRAFFIC

CACHE-TO-CACHE TRANSFER IN DIRECTORY-BASED PROTOCOLS



INDIRECTION PROBLEM: LATENCY/TRAFFIC

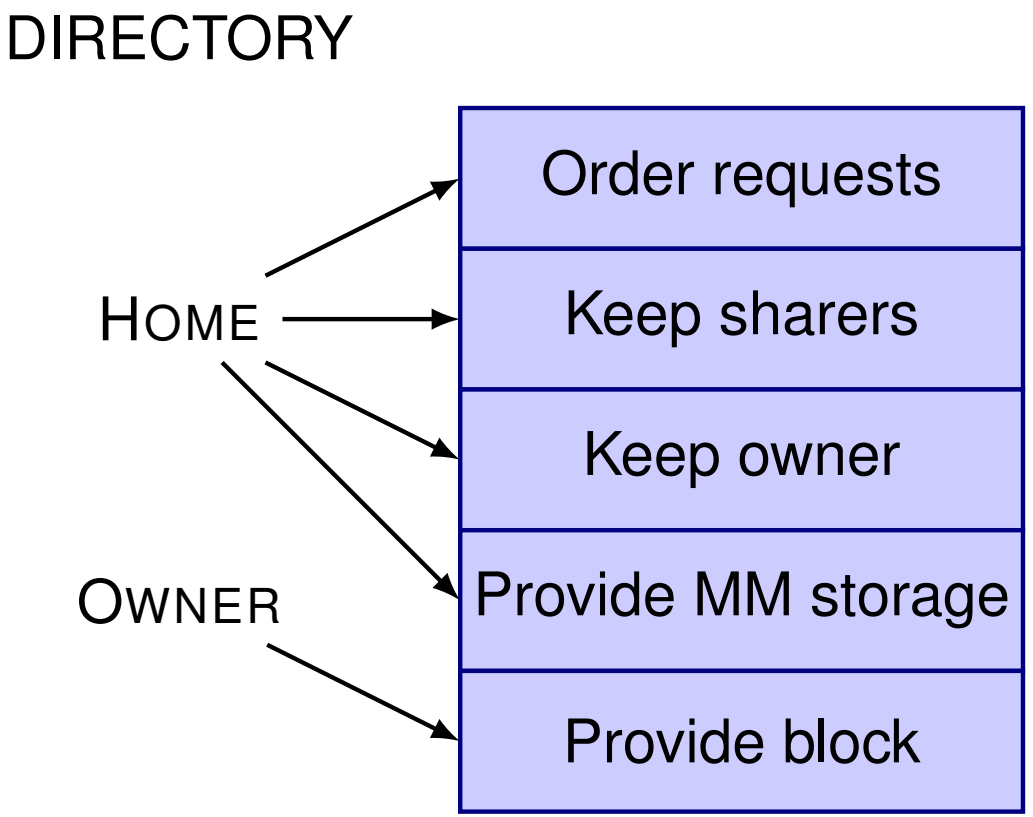
CACHE-TO-CACHE TRANSFER IN DIRECTORY-BASED PROTOCOLS



FIRST IDEA: what if the requester sends directly the data to the owner? ADVISOR: Already done (by me...)

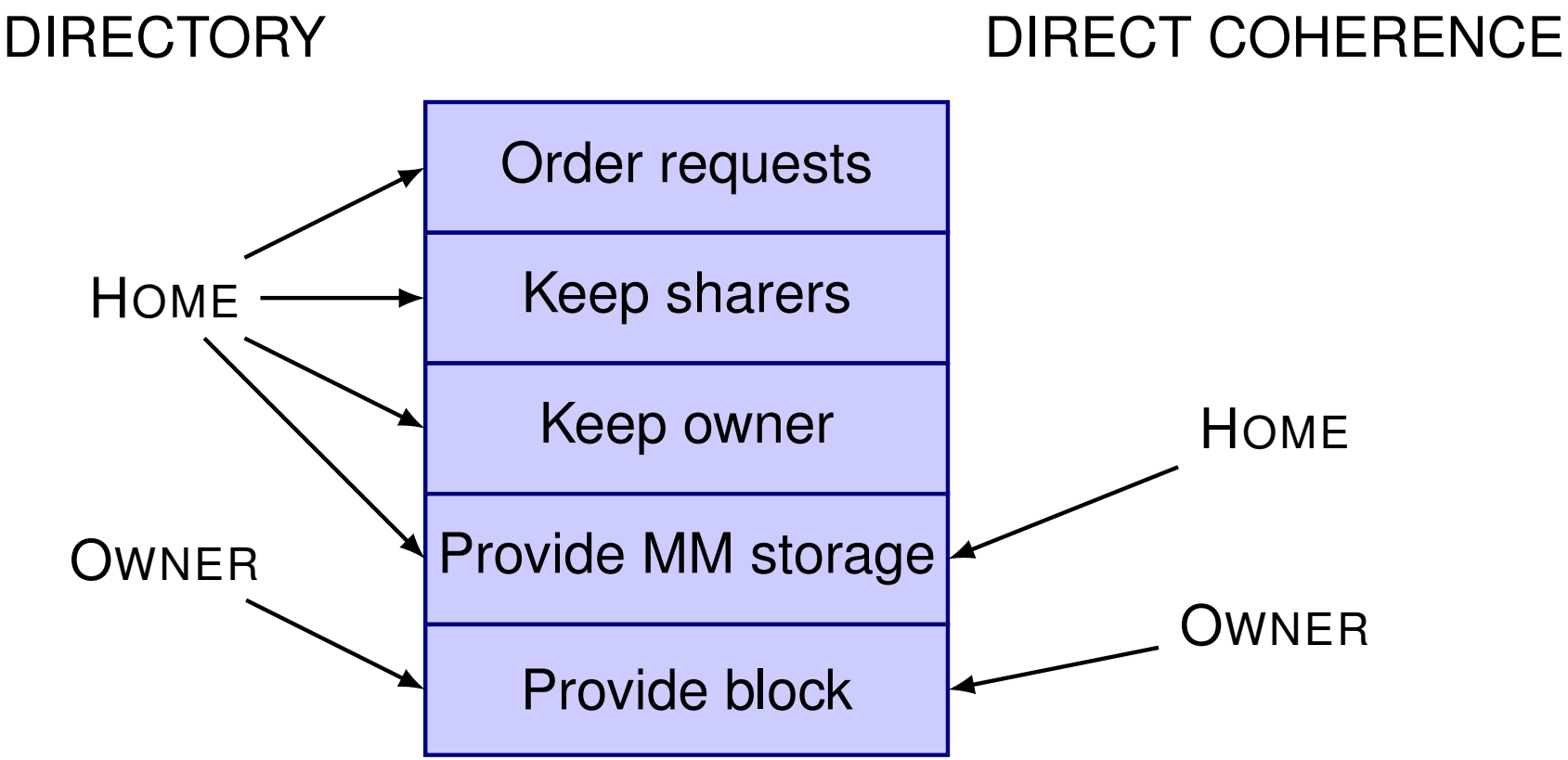
DIRECT COHERENCE: THE ROLES

Ros, Acacio & García [IPDPS'08]



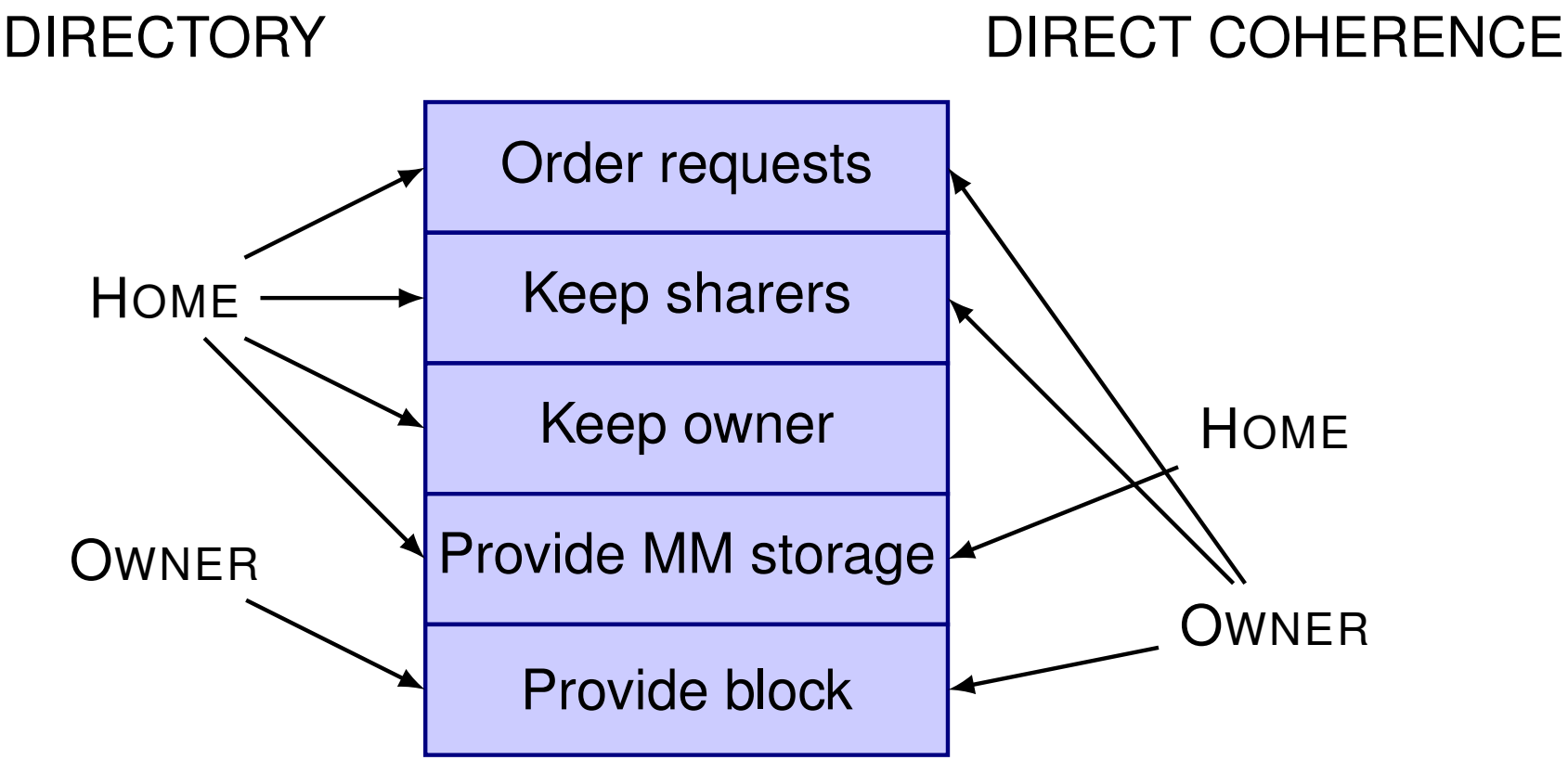
DIRECT COHERENCE: THE ROLES

Ros, Acacio & García [IPDPS'08]



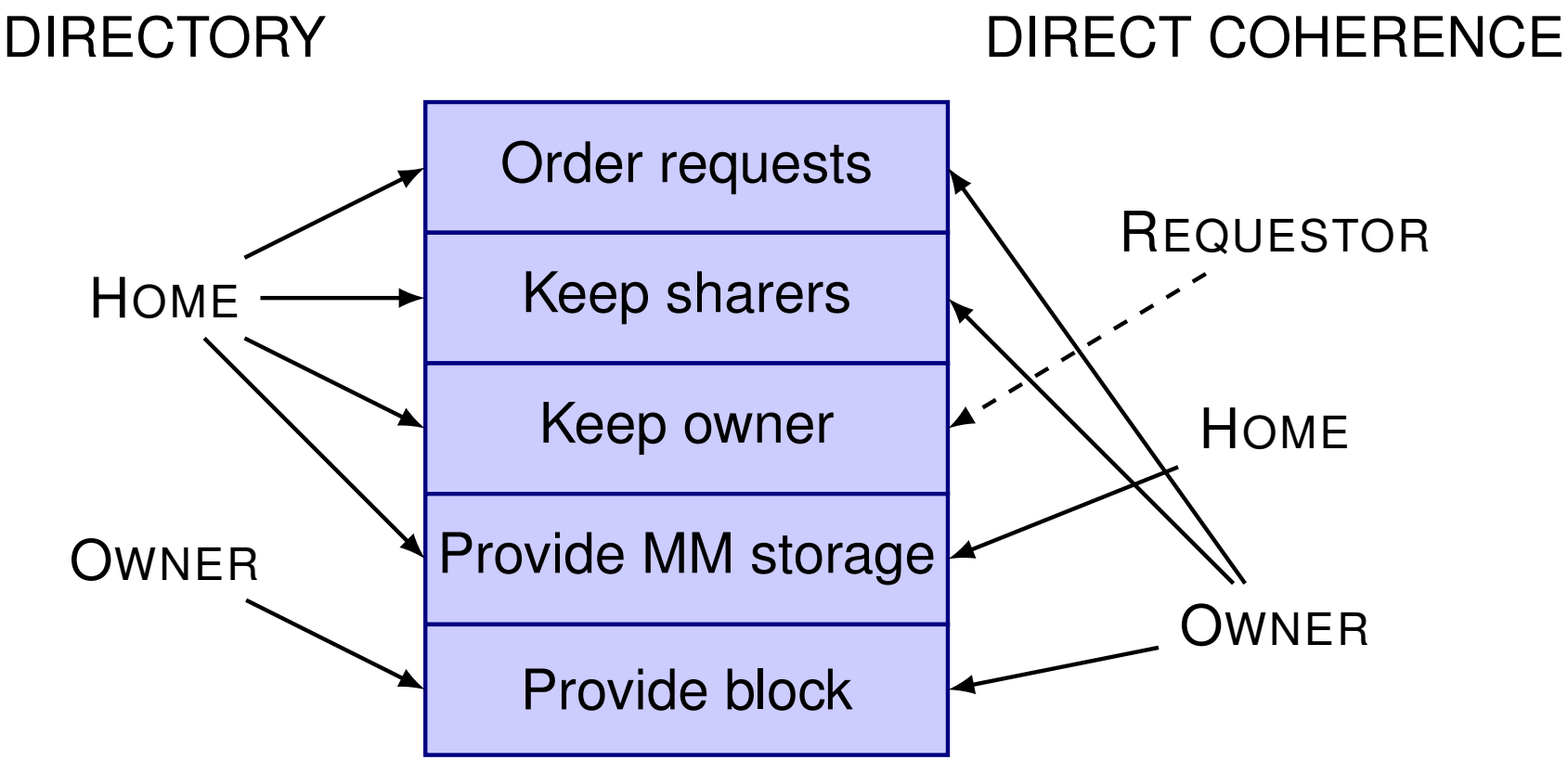
DIRECT COHERENCE: THE ROLES

Ros, Acacio & García [IPDPS'08]



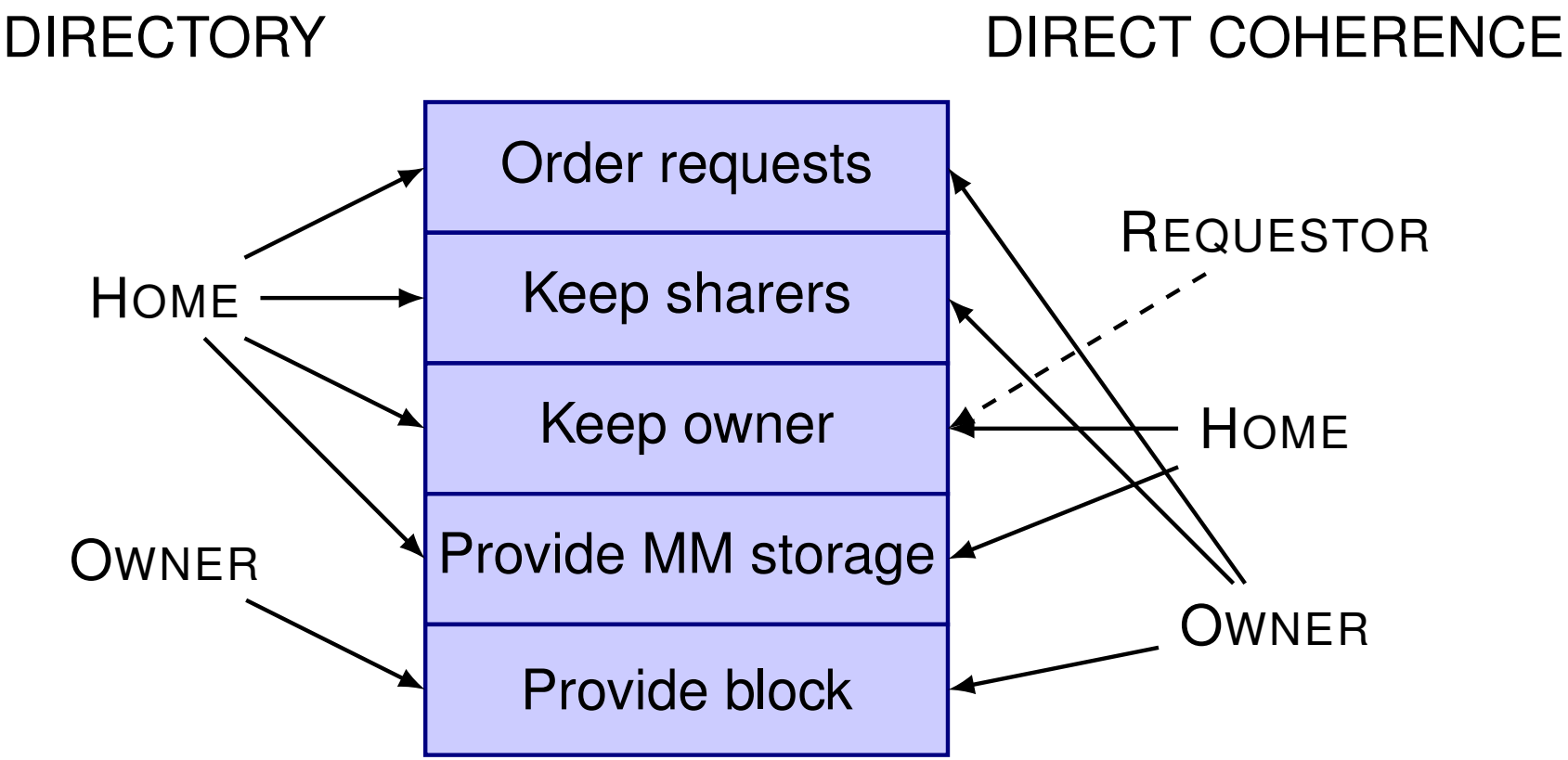
DIRECT COHERENCE: THE ROLES

Ros, Acacio & García [IPDPS'08]



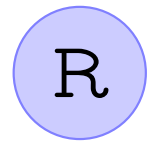
DIRECT COHERENCE: THE ROLES

Ros, Acacio & García [IPDPS'08]



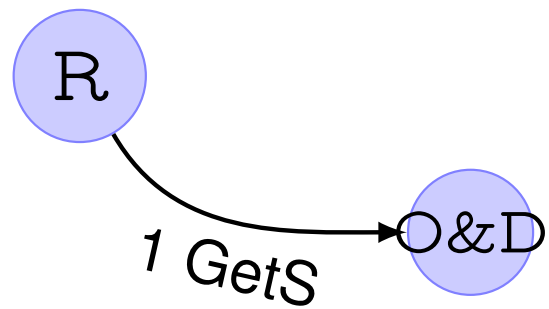
DIRECT COHERENCE IN ACTION
CACHE-TO-CACHE READ MISS

DIRECT COHERENCE



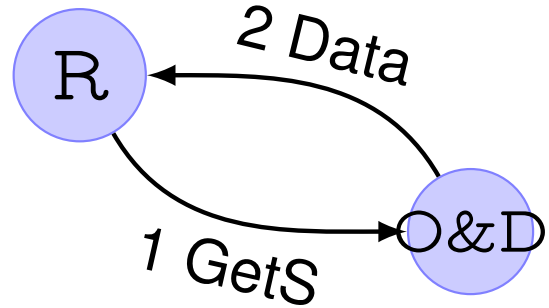
Direct Coherence in action
Cache-to-Cache Read Miss

Direct Coherence



Direct Coherence in action
Cache-to-Cache Read Miss

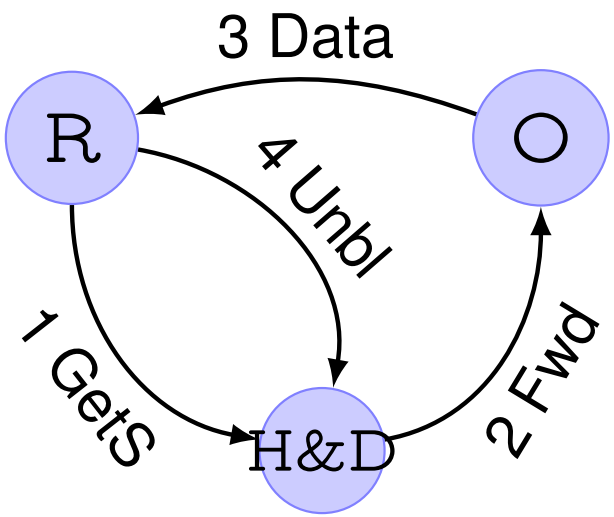
Direct Coherence



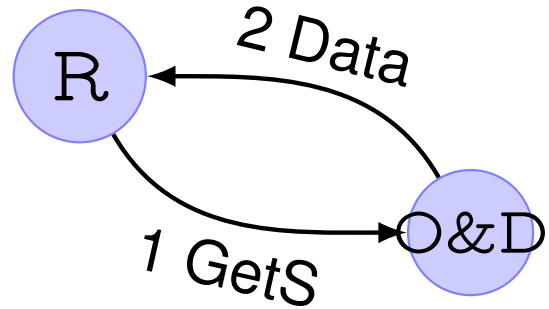
Direct Coherence in action

Cache-to-Cache Read Miss

Directory



Direct Coherence



- The critical path of the miss is reduced from three to two hops.
- The number of coherence messages is halved.
- The waiting time at the home tile is removed.

THINKING ABOUT INVALIDATIONS

ME: What if stores do not wait for invalidations to *perform*?

ADVISOR: It would break consistency.

ME: Right!

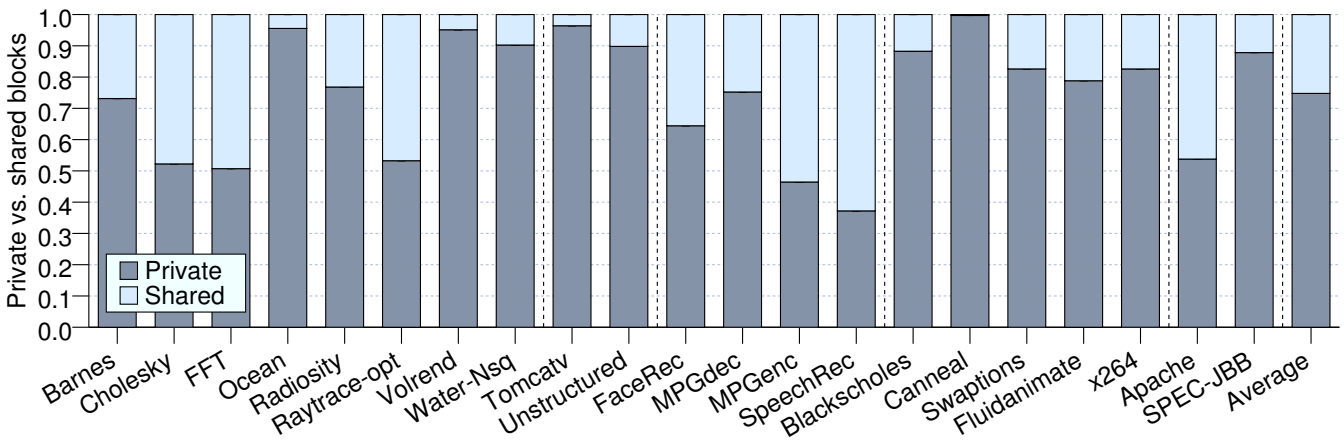
- But... Importance of questioning always the *why*

REDUCING DIRECTORY STORAGE

COHERENCE DEACTIVATION

Cuesta, Ros, Gómez, Robles & Duato [ISCA'11]

- Directories track lines stored in private caches to support efficient invalidation
  - But is it necessary to track all referenced blocks?
  - Private blocks will never be incoherent!
    - 75% of referenced blocks (on average)



- If we do not keep directory information for private blocks we can save directory storage.

Coherence Deactivation

The mechanism

- Classifies memory blocks into private and shared (how?)
  - It can be seen as a two-level classification
- Deactivates the coherence protocol for private blocks
  - i.e., do not keep track of them
- Blocks are classified at page granularity
- The operating system detects when a page (initially considered private) becomes shared (minimal OS overhead).
  - Performed upon TLB misses: state stored in the page table
  - A coherence recovery mechanism is necessary to restore block's coherence status.
- Collaboration between hardware and operating system
- Inspired projects and lots of works

MEMORY CONSISTENCY MODELS

LET'S SWITCH TO MEMORY CONSISTENCY

DEKKER'S ALGORITHM

```
/* Initially X = Y = 0 */

X = 1;          |  Y = 1;
$r0 = Y;        |  $r1 = X;
```

- Can we get $r0==0, $r1==0?

LET'S SWITCH TO MEMORY CONSISTENCY

DEKKER'S ALGORITHM

```
/* Initially X = Y = 0 */

X = 1;          |  Y = 1;
$r0 = Y;        |  $r1 = X;
```

- Can we get $r0==0, $r1==0?
- Only if in a thread the load performs before the store in the same thread

Memory Consistency Model

- To argue about the correctness of a program, it is necessary to define a memory consistency model (or consistency model)
- The consistency model is the contract between the programmer and the system
  - The programmer knows which results he can expect
  - The system know how much the program can be optimized

Memory Consistency Model

- To argue about the correctness of a program, it is necessary to define a memory consistency model (or consistency model)
- The consistency model is the contract between the programmer and the system
  - The programmer knows which results he can expect
  - The system know how much the program can be optimized

Definition

The consistency model is a specification of the behaviour of multithreaded programs executing under shared memory

Memory Consistency Model

- To argue about the correctness of a program, it is necessary to define a memory consistency model (or consistency model)
- The consistency model is the contract between the programmer and the system
  - The programmer knows which results he can expect
  - The system know how much the program can be optimized

Definition

The consistency model is a specification of the behaviour of multithreaded programs executing under shared memory

- In particular, the consistency model specifies the values returned by every load and the final status of the memory when the program finishes executing

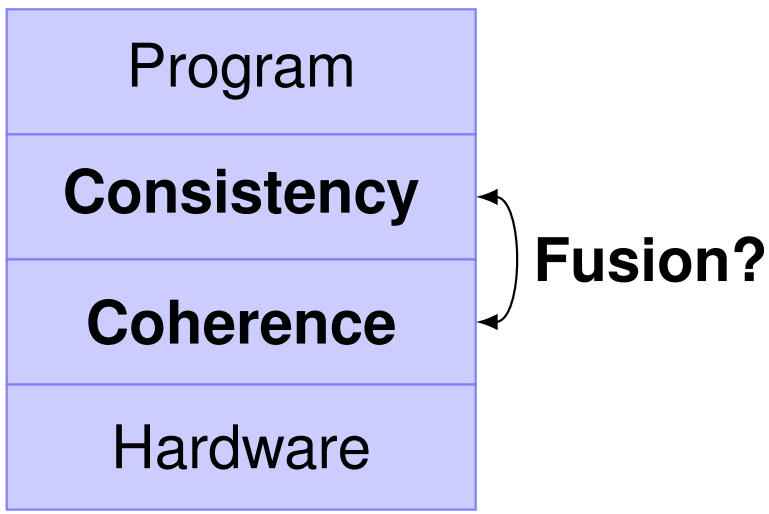
Consistency vs. Coherence

- Memory consistency model: defines correct program behavior
- Cache coherence protocol: eases the implementation of a consistency model
  - Makes caches in a shared memory multicore functionally invisible (timing can be inferred)



Consistency vs. Coherence

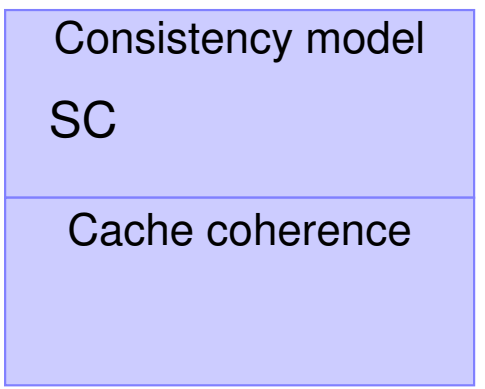
- Memory consistency model: defines correct program behavior
- Cache coherence protocol: eases the implementation of a consistency model
  - Makes caches in a shared memory multicore functionally invisible (timing can be inferred)



But what if we break the coherence-consistency interface?

BREAKING THE LAYERS

- Cache coherence serves as a black box to support strict consistency models: e.g., Sequential Consistency (SC)



BREAKING THE LAYERS

- Cache coherence serves as a black box to support strict consistency models: e.g., Sequential Consistency (SC)
  - Single-writer-multiple-readers (SWMR) invariant
  - Invalidation/update of the copies on every write
  - Large amount of traffic ⇒ increases energy consumption

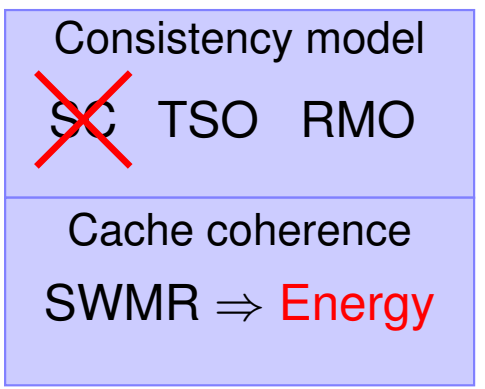
Consistency model
SC
Cache coherence
SWMR ⇒ Energy

BREAKING THE LAYERS

- Cache coherence serves as a black box to support strict consistency models: e.g., Sequential Consistency (SC)
  - Single-writer-multiple-readers (SWMR) invariant
  - Invalidation/update of the copies on every write
  - Large amount of traffic ⇒ increases energy consumption

OBSERVATION

Most processors offer consistency models weaker than SC



BREAKING THE LAYERS

- Cache coherence serves as a black box to support strict consistency models: e.g., Sequential Consistency (SC)
  - Single-writer-multiple-readers (SWMR) invariant
  - Invalidation/update of the copies on every write
  - Large amount of traffic $\Rightarrow$ increases energy consumption

OBSERVATION

Most processors offer consistency models weaker than SC

- Why implement protocols that provide more functionality than necessary?



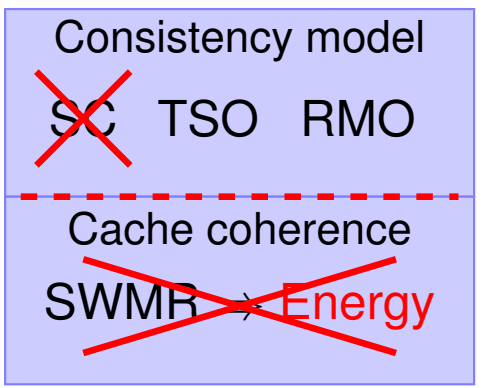
BREAKING THE LAYERS

- Cache coherence serves as a black box to support strict consistency models: e.g., Sequential Consistency (SC)
  - Single-writer-multiple-readers (SWMR) invariant
  - Invalidation/update of the copies on every write
  - Large amount of traffic $\Rightarrow$ increases energy consumption

OBSERVATION

Most processors offer consistency models weaker than SC

- Why implement protocols that provide more functionality than necessary?
- Protocol as a black box?
  - Break the layer between the consistency model and the coherence protocol!



TOWARD SIMPLE REQUEST-RESPONSE COHERENCE

- Write-through protocols are simple (GPUs)
  - Only Valid and Invalid states

TOWARD SIMPLE REQUEST-RESPONSE COHERENCE

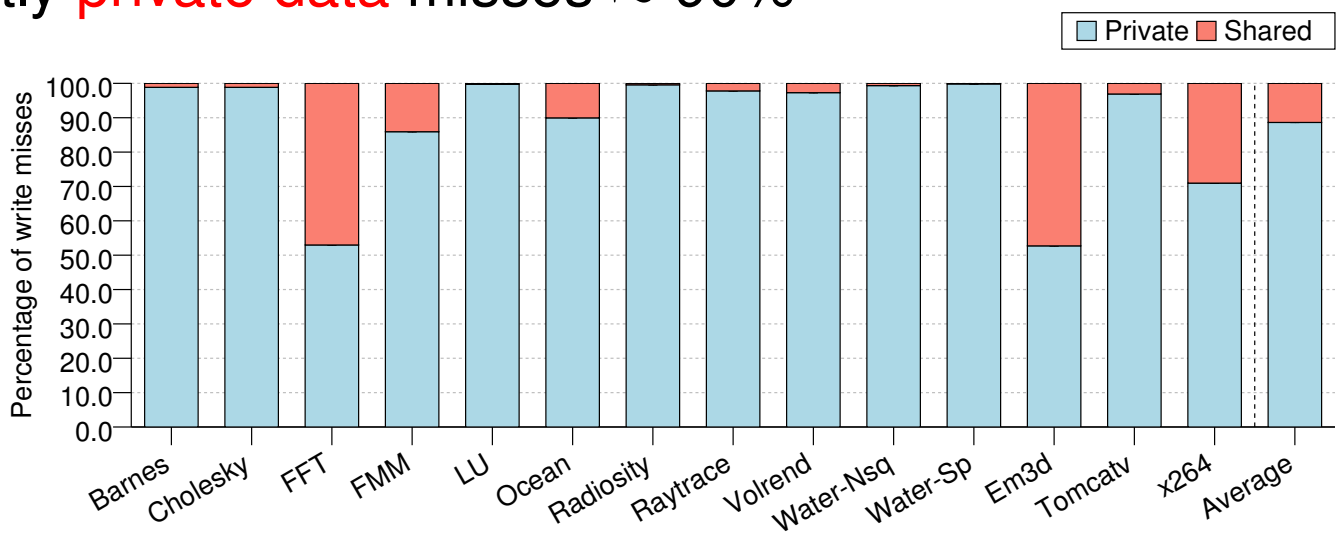
- Write-through protocols are simple (GPUs)
  - Only Valid and Invalid states
  - But they are not efficient because of write misses
- Which write misses?

TOWARD SIMPLE REQUEST-RESPONSE COHERENCE

- Write-through protocols are simple (GPUs)
  - Only Valid and Invalid states
  - But they are not efficient because of write misses
- Which write misses?
  - Private data in a write-back policy
    - → evicted due to capacity/conflict misses
  - Shared data in a write-back policy
    - → evicted due to capacity/conflict/coherence misses

TOWARD SIMPLE REQUEST-RESPONSE COHERENCE

- Write-through protocols are simple (GPUs)
  - Only Valid and Invalid states
  - But they are not efficient because of write misses
- Which write misses?
  - Private data in a write-back policy
    - → evicted due to capacity/conflict misses
  - Shared data in a write-back policy
    - → evicted due to capacity/conflict/coherence misses
- Mostly private data misses ≈ 90%



VIPS: Dynamic write policy

Ros & Kaxiras [PACT'12]

Dynamic write policy in the L1s (private caches, in general)

- Write-back for **P**rivate blocks
  - Simple (no coherence required) as in uniprocessors
  - Efficient $\rightarrow$ no extra misses
- Write-through for **S**hared blocks
  - Simple (only two states, **VI**)
  - Efficient $\rightarrow$ coherence misses

Requires a private/shared classification!

VIPS: **V**alid/**I**nvalid **P**rivate/**S**hared

VIPS: SELF-INVALIDATION

- Supports sequential consistency for every application
  - Same consistency model as the more complex MESI
- But we still have invalidations and directory blocking...
  - How can we remove this restriction? $\rightarrow$ Incoherence...

VIPS: SELF-INVALIDATION

- Supports sequential consistency for every application
  - Same consistency model as the more complex MESI
- But we still have invalidations and directory blocking...
  - How can we remove this restriction? → Incoherence...

- SOLUTION: sequential consistency for DRF programs
- How? Synchronization exposed to the protocol

EXAMPLE OF DRF CODE

```
X = 1;              | WAIT(cond);
SIGNAL(cond);       | $r1 = X;
```

VIPS: SELF-INVALIDATION

- Supports sequential consistency for every application
  - Same consistency model as the more complex MESI
- But we still have invalidations and directory blocking...
  - How can we remove this restriction? → Incoherence...

- SOLUTION: sequential consistency for DRF programs
- How? Synchronization exposed to the protocol
- Release: SELF-DOWNGRADE
  ⇒ Write-through dirty blocks

EXAMPLE OF DRF CODE

SD	X = 1;	WAIT(cond);
	SIGNAL(cond);	$r1 = X;

VIPS: SELF-INVALIDATION

- Supports sequential consistency for every application
  - Same consistency model as the more complex MESI
- But we still have invalidations and directory blocking...
  - How can we remove this restriction? → Incoherence...

- SOLUTION: sequential consistency for DRF programs
- How? Synchronization exposed to the protocol

- Release: SELF-DOWNGRADE
  ⇒ Write-through dirty blocks
- Acquire: SELF-INVALIDATION
  ⇒ Empty the cache

EXAMPLE OF DRF CODE

SD

```
X = 1;          | WAIT(cond);
SIGNAL(cond);   | $r1 = X;
```

SI

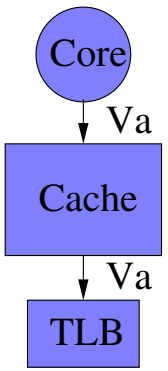
VIPS LEGACY

- Allows supporting virtual (VIVT) L1 caches

  Kaxiras & Ros [ISCA'13]

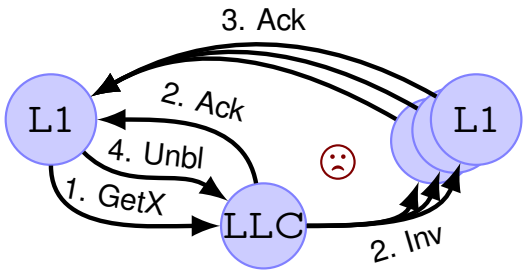
- Allows efficient syncronization (callbacks)

  Ros & Kaxiras [ISCA'15]



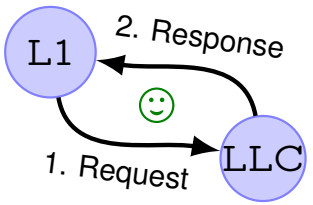
SC VERSUS SC-FOR-DRF COHERENCE

SC



☺ Works for all codes

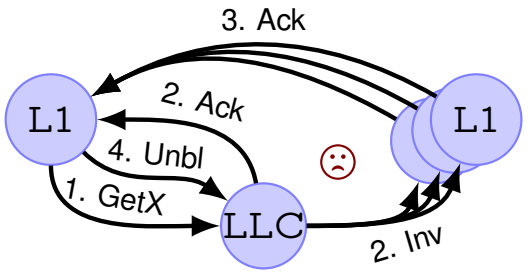
SC-FOR-DRF



☹ Only works for DRF codes

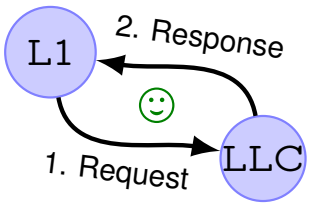
SC VERSUS SC-FOR-DRF COHERENCE

SC



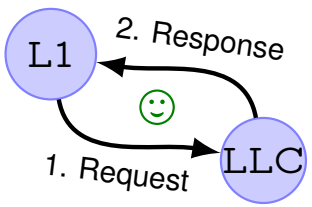
☺ Works for all codes

SC-FOR-DRF



☹ Only works for DRF codes

RACER (TOTAL-STORE-ORDER – TSO)



☺ Works for all (TSO) codes

First efficient, request-response protocol for all codes

RACER AT A GLANCE

Ros & Kaxiras [MICRO'16]

- A novel way of supporting TSO consistency
  - $\Rightarrow$ At the cache coherence protocol level

RACER AT A GLANCE

Ros & Kaxiras [MICRO'16]

- A novel way of supporting TSO consistency
  - ⇒ At the cache coherence protocol level
- We start with a very simple request-response protocol
  - ⇒ Order enforced on SI & SD

RACER AT A GLANCE

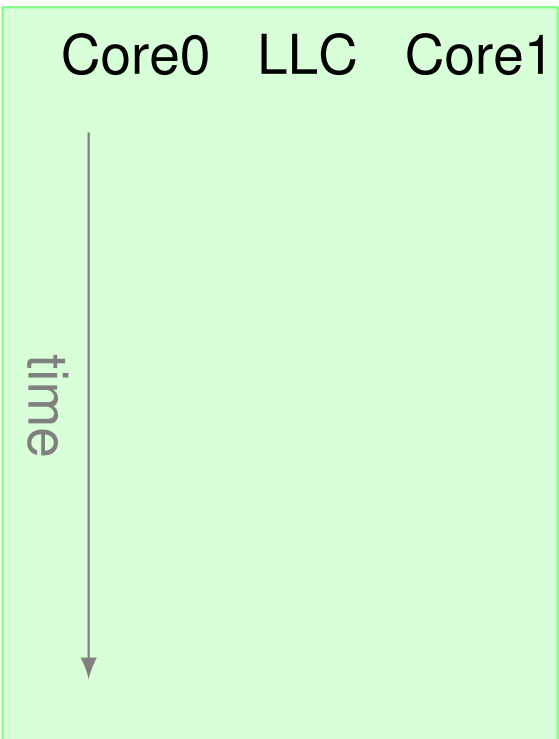
Ros & Kaxiras [MICRO'16]

- A novel way of supporting TSO consistency
  - ⇒ At the cache coherence protocol level
- We start with a very simple request-response protocol
  - ⇒ Order enforced on SI & SD
- When it is necessary to enforce order?
  - ⇒ In SC-for-DRF conservatively on synchronization

RACER AT A GLANCE

Ros & Kaxiras [MICRO'16]

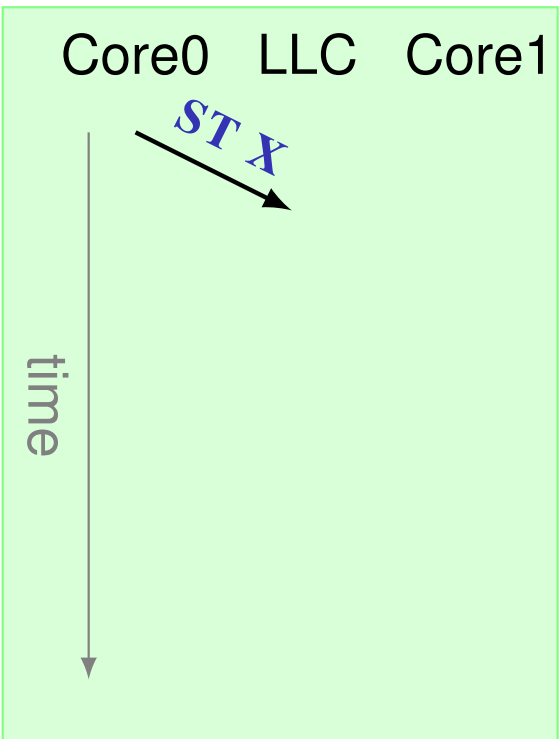
- A novel way of supporting TSO consistency
  - ⇒ At the cache coherence protocol level
- We start with a very simple request-response protocol
  - ⇒ Order enforced on SI & SD
- When it is necessary to enforce order?
  - ⇒ In SC-for-DRF conservatively on synchronization
  - ⇒ In RACER only when it is possible to see a reordering
  - ⇒ On actual (read-after-write) RAW races



RACER AT A GLANCE

Ros & Kaxiras [MICRO'16]

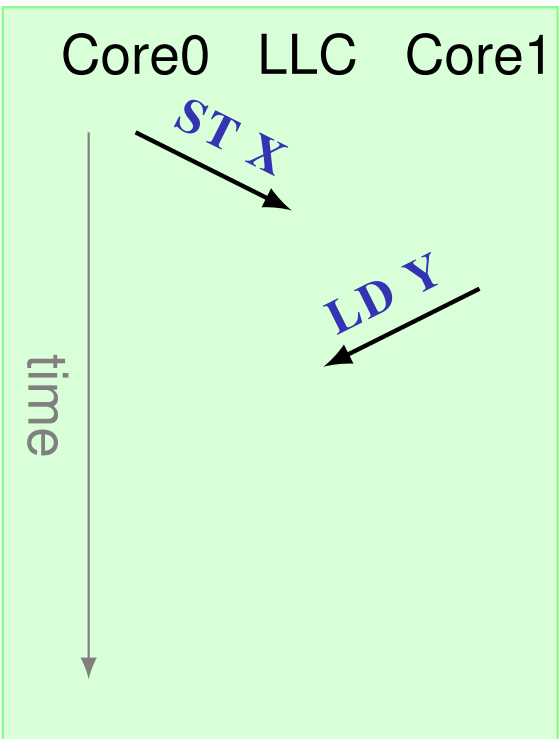
- A novel way of supporting TSO consistency
  - ⇒ At the cache coherence protocol level
- We start with a very simple request-response protocol
  - ⇒ Order enforced on SI & SD
- When it is necessary to enforce order?
  - ⇒ In SC-for-DRF conservatively on synchronization
  - ⇒ In RACER only when it is possible to see a reordering
  - ⇒ On actual (read-after-write) RAW races



RACER AT A GLANCE

Ros & Kaxiras [MICRO'16]

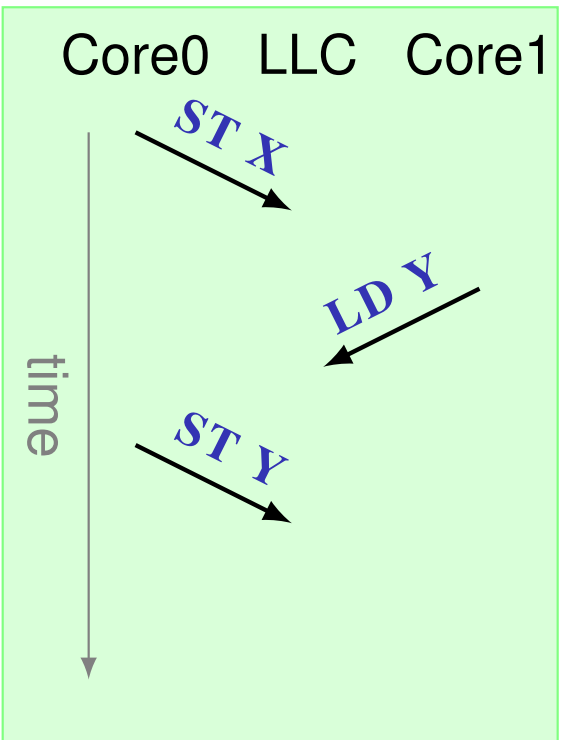
- A novel way of supporting TSO consistency
  - ⇒ At the cache coherence protocol level
- We start with a very simple request-response protocol
  - ⇒ Order enforced on SI & SD
- When it is necessary to enforce order?
  - ⇒ In SC-for-DRF conservatively on synchronization
  - ⇒ In RACER only when it is possible to see a reordering
  - ⇒ On actual (read-after-write) RAW races



RACER AT A GLANCE

Ros & Kaxiras [MICRO'16]

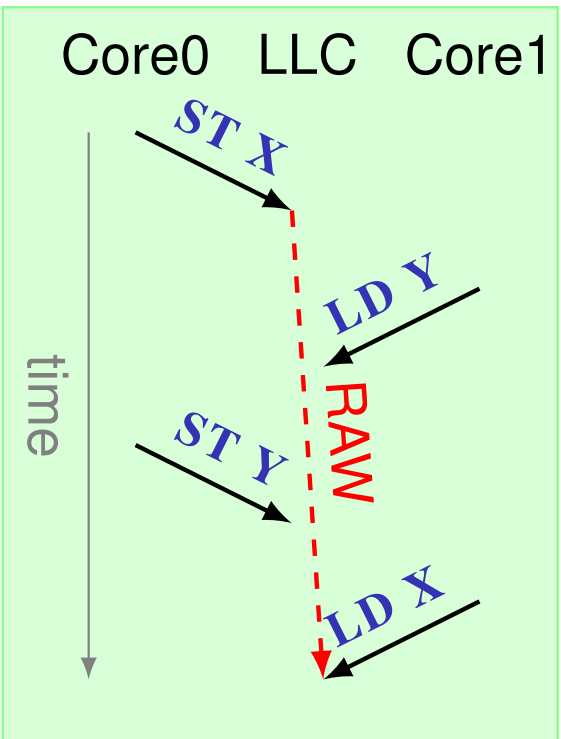
- A novel way of supporting TSO consistency
  - ⇒ At the cache coherence protocol level
- We start with a very simple request-response protocol
  - ⇒ Order enforced on SI & SD
- When it is necessary to enforce order?
  - ⇒ In SC-for-DRF conservatively on synchronization
  - ⇒ In RACER only when it is possible to see a reordering
  - ⇒ On actual (read-after-write) RAW races



RACER AT A GLANCE

Ros & Kaxiras [MICRO'16]

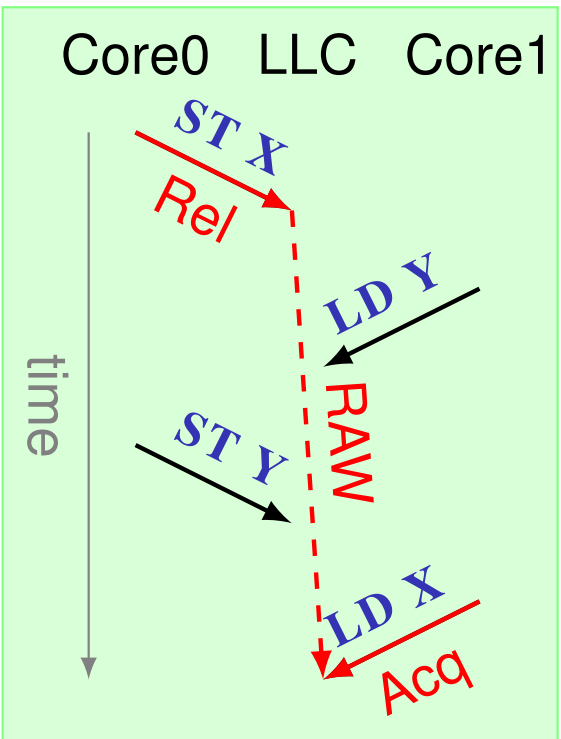
- A novel way of supporting TSO consistency
  - ⇒ At the cache coherence protocol level
- We start with a very simple request-response protocol
  - ⇒ Order enforced on SI & SD
- When it is necessary to enforce order?
  - ⇒ In SC-for-DRF conservatively on synchronization
  - ⇒ In RACER only when it is possible to see a reordering
  - ⇒ On actual (read-after-write) RAW races



RACER AT A GLANCE

Ros & Kaxiras [MICRO'16]

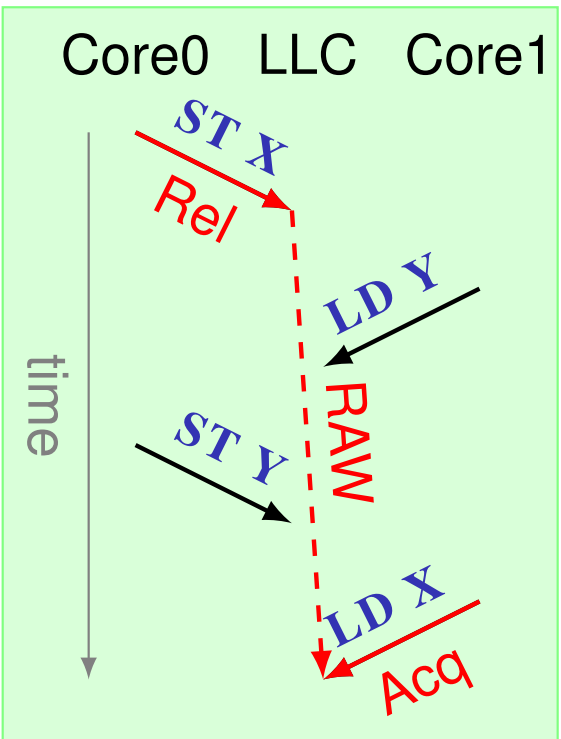
- A novel way of supporting TSO consistency
  - ⇒ At the cache coherence protocol level
- We start with a very simple request-response protocol
  - ⇒ Order enforced on SI & SD
- When it is necessary to enforce order?
  - ⇒ In SC-for-DRF conservatively on synchronization
  - ⇒ In RACER only when it is possible to see a reordering
  - ⇒ On actual (read-after-write) RAW races



RACER AT A GLANCE

Ros & Kaxiras [MICRO'16]

- A novel way of supporting TSO consistency
  - ⇒ At the cache coherence protocol level
- We start with a very simple request-response protocol
  - ⇒ Order enforced on SI & SD
  - When it is necessary to enforce order?
    - ⇒ In SC-for-DRF conservatively on synchronization
    - ⇒ In RACER only when it is possible to see a reordering
    - ⇒ On actual (read-after-write) RAW races
  - Consistency only enforced for shared data



What we learned

- Cache coherence is not strictly necessary for implementing consistency models
- Merging the coherence and consistency layers is challenging
- ... but a topic worth to explore!

NON-SPECULATIVE EXECUTION

TOTAL STORE ORDER (TSO) CONSISTENCY

- Respected orders:

load→load load→store store→load store→store

INITIALLY X = Y = 0

```
lx: $r0 = X;  |  sy: Y = 1;
ly: $r1 = Y;  |  sx: X = 1;
```

Total Store Order (TSO) Consistency

- Respected orders:

load→load load→store store→load store→store

Initially X = Y = 0

```
lx: $r0 = X;  |  sy: Y = 1;
ly: $r1 = Y;  |  sx: X = 1;
```

Six possible interleavings and values for $r0 and $r1

```
lx          |  lx          |  lx          |       sy     |       sy     |       sy
ly          |       sy     |       sy     |  lx          |  lx          |       sx
     sy     |  ly          |       sx     |  ly          |       sx     |  lx
     sx     |       sx     |  ly          |       sx     |  ly          |  ly

  (0,0)     |    (0,1)     |    (0,1)     |    (0,1)     |    (0,1)     |    (1,1)
```

- (1,0) is not possible under load→load & store→store

MEMORY CONSISTENCY SPECULATION

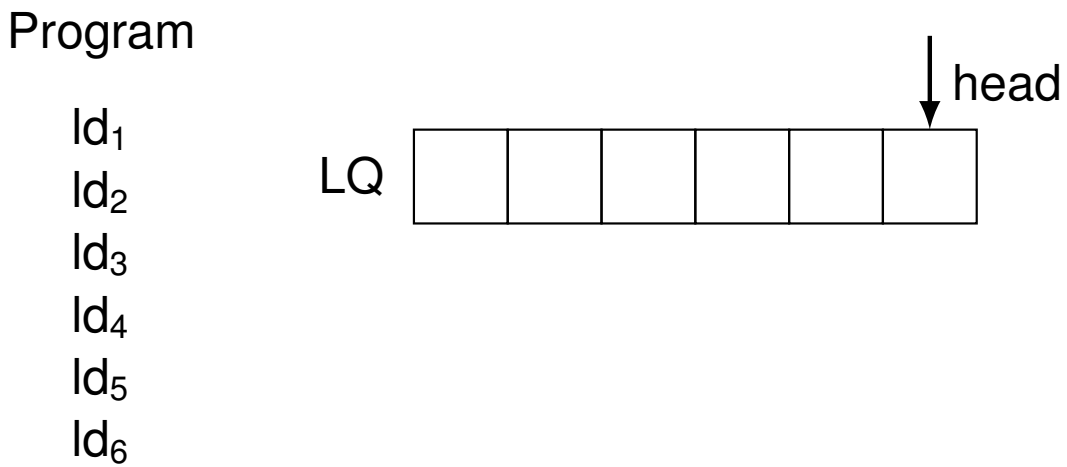
- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism

MEMORY CONSISTENCY SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism

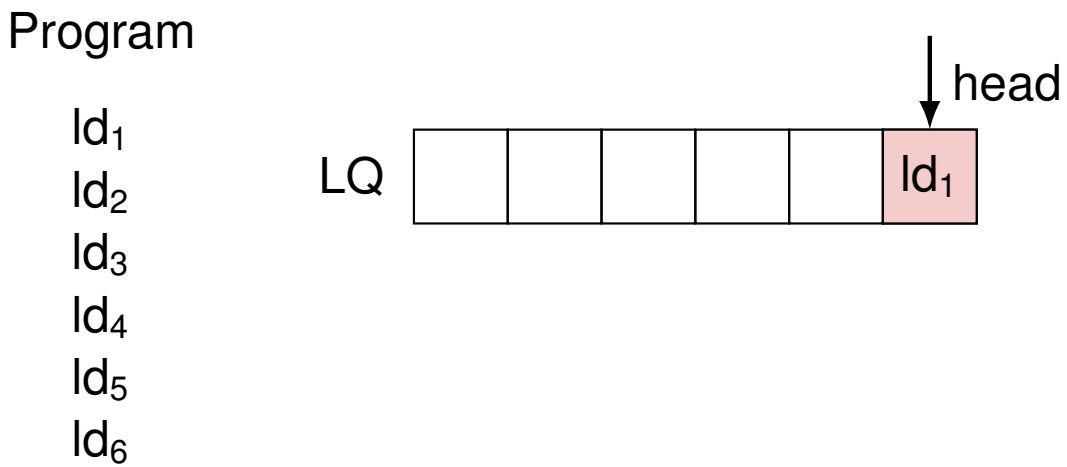
MEMORY CONSISTENCY SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism
- They can execute loads out-of-order, so loads can be reordered



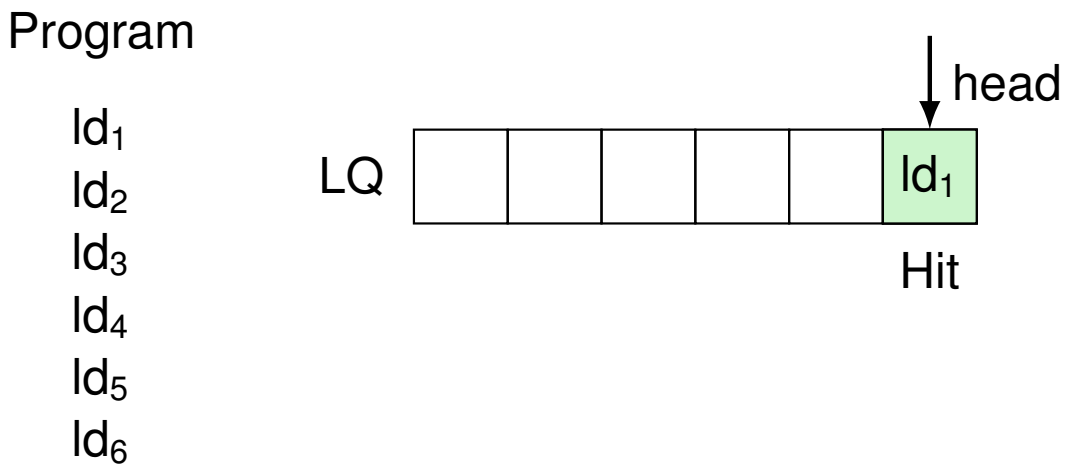
MEMORY CONSISTENCY SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism
- They can execute loads out-of-order, so loads can be reordered



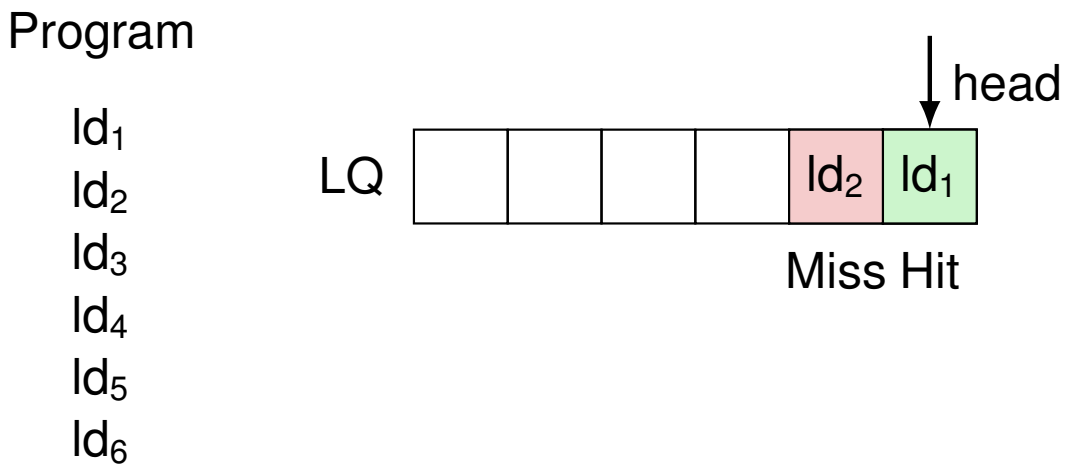
MEMORY CONSISTENCY SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism
- They can execute loads out-of-order, so loads can be reordered



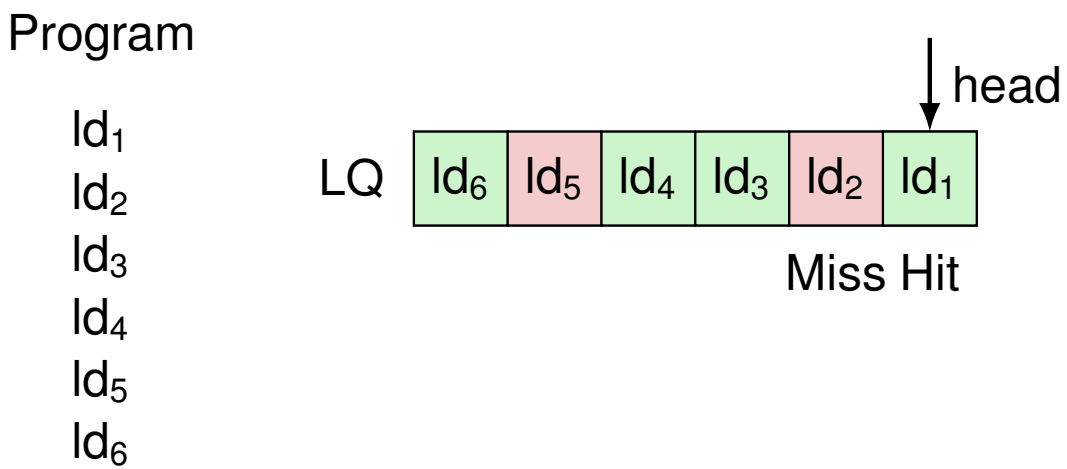
MEMORY CONSISTENCY SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism
- They can execute loads out-of-order, so loads can be reordered



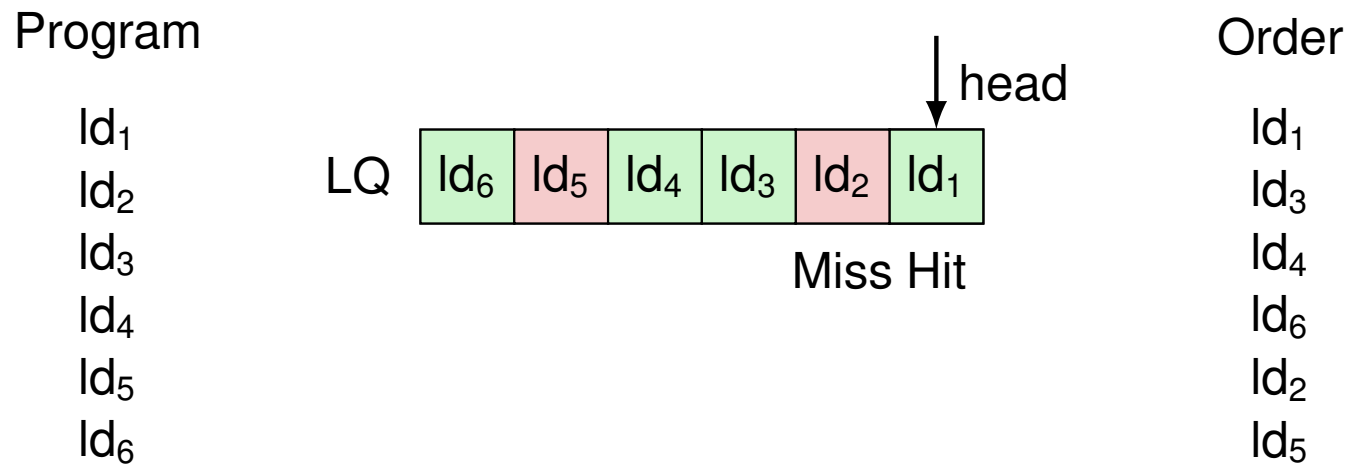
MEMORY CONSISTENCY SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism
- They can execute loads out-of-order, so loads can be reordered



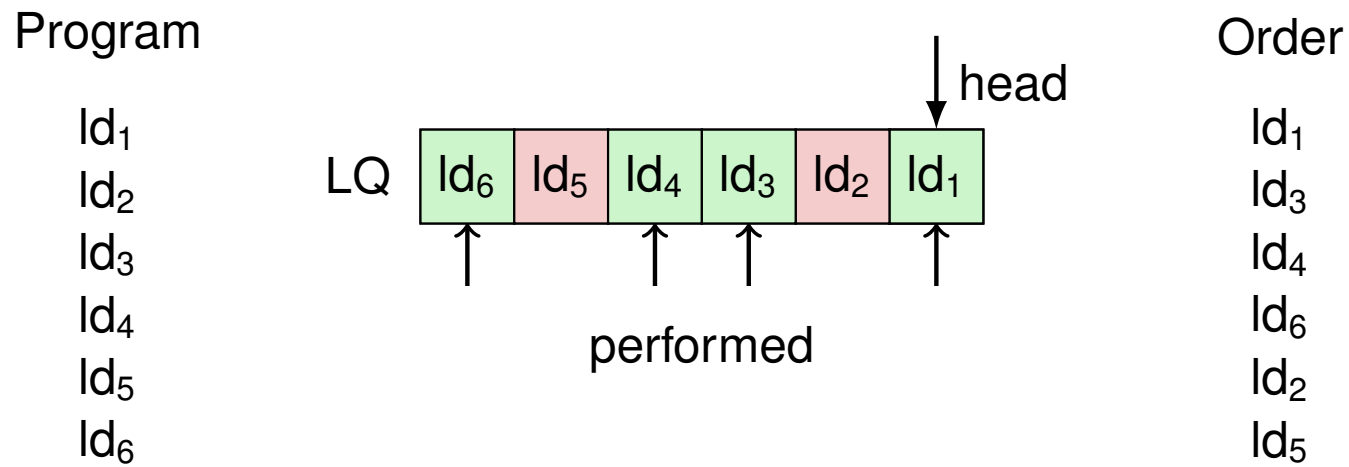
MEMORY CONSISTENCY SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism
- They can execute loads out-of-order, so loads can be reordered
- This could violate the load→load rule (e.g. hit-miss)



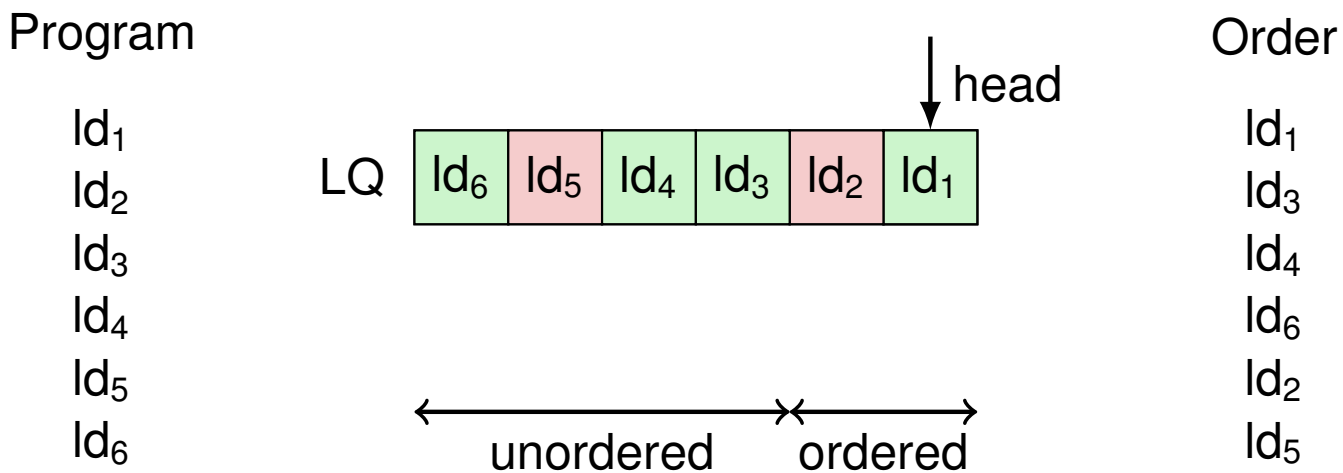
MEMORY CONSISTENCY SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism
- They can execute loads out-of-order, so loads can be reordered
- This could violate the load→load rule (e.g. hit-miss)



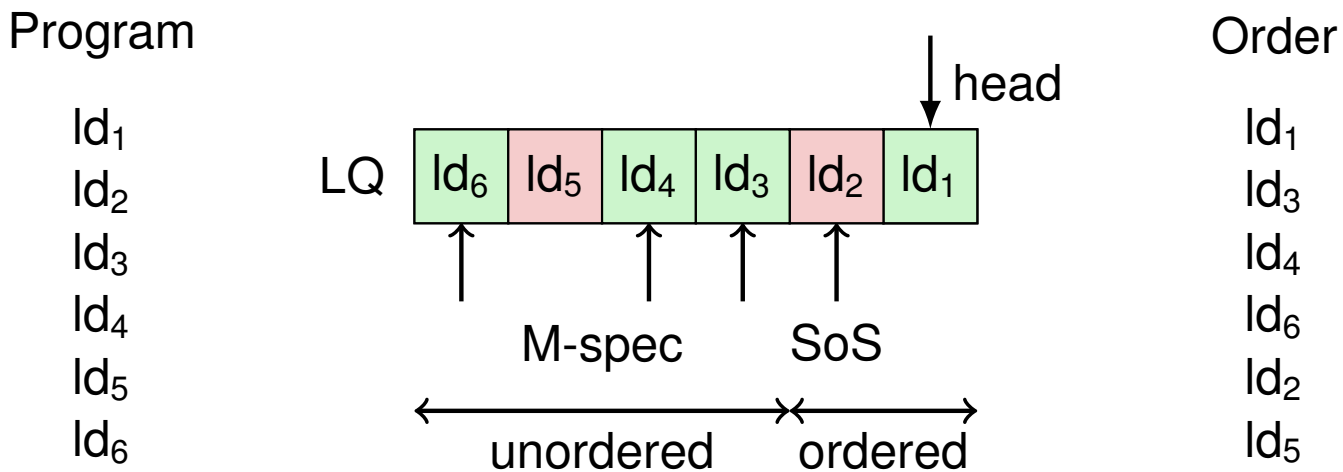
MEMORY CONSISTENCY SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism
- They can execute loads out-of-order, so loads can be reordered
- This could violate the load→load rule (e.g. hit-miss)



MEMORY CONSISTENCY SPECULATION

- High-performance multicores perform many memory operations simultaneously
  - Memory-level paralelism
- They can execute loads out-of-order, so loads can be reordered
- This could violate the load→load rule (e.g. hit-miss)



BREAKING LOAD→LOAD

INITIALLY X = Y = 0

```
lx: $r0 = X;  |  sy: Y = 1;
ly: $r1 = Y;  |  sx: X = 1;
```

SIX POSSIBLE INTERLEAVINGS BREAKING LOAD→LOAD

ly	ly	ly	sy	sy	sy
lx	sy	sy	ly	ly	sx
sy	lx	sx	lx	sx	ly
sx	sx	lx	sx	lx	lx
(0,0)	(0,0)	(1,0)	(0,1)	(1,1)	(1,1)

- (1,0) is possible when relaxing load→load

SQUASH OR WAIT?

- Current multicores avoid non-valid results by
  - Interacting with the coherence protocol

```
ly
      sy
      sx
lx

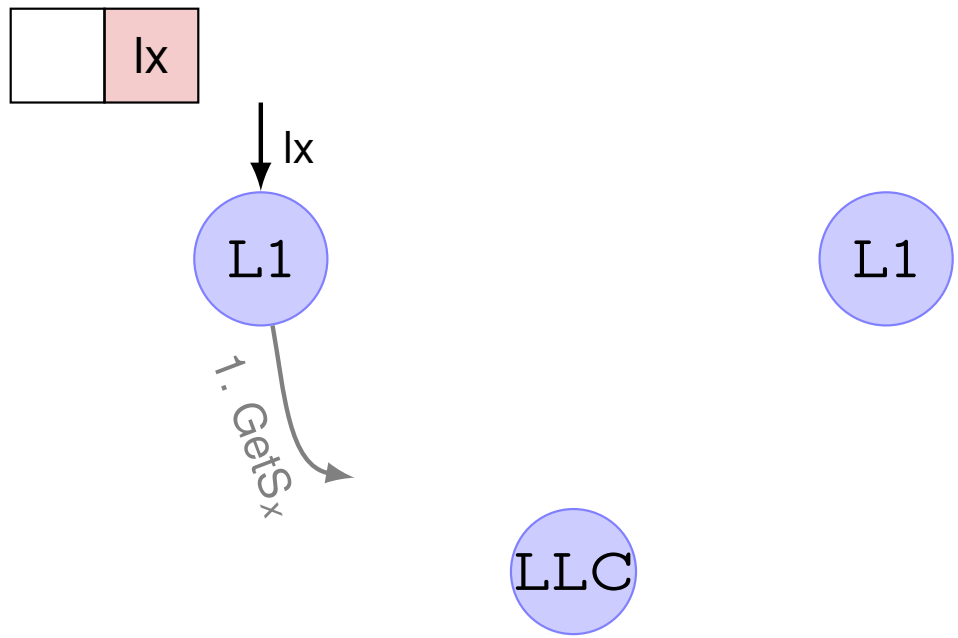
 (1,0)
```

SQUASH OR WAIT?

- Current multicores avoid non-valid results by
  - Interacting with the coherence protocol

```
ly
        sy
        sx
lx

  (1,0)
```

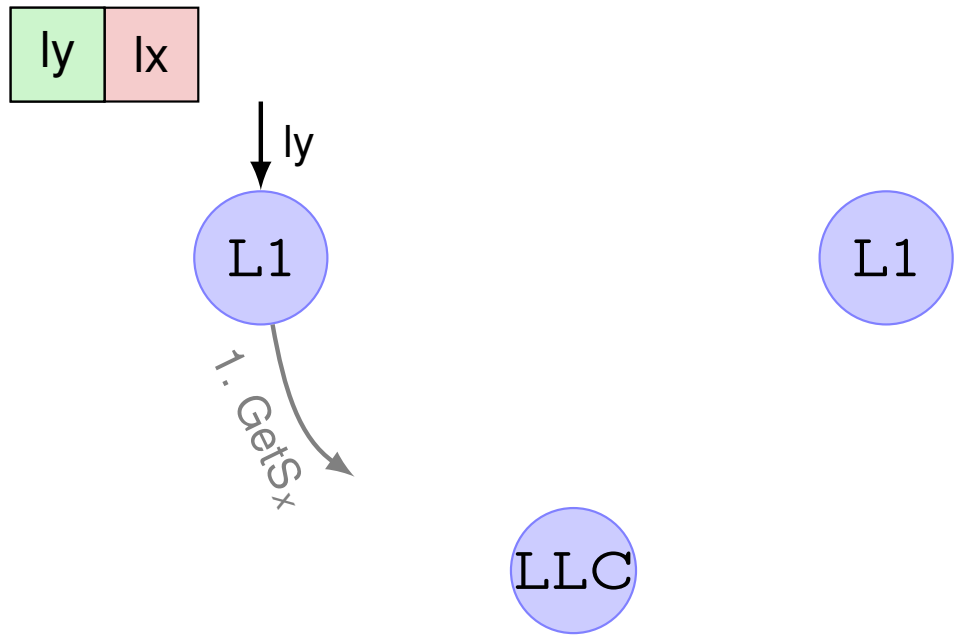


SQUASH OR WAIT?

- Current multicores avoid non-valid results by
  - Interacting with the coherence protocol

```
ly
      sy
      sx
lx

 (1,0)
```

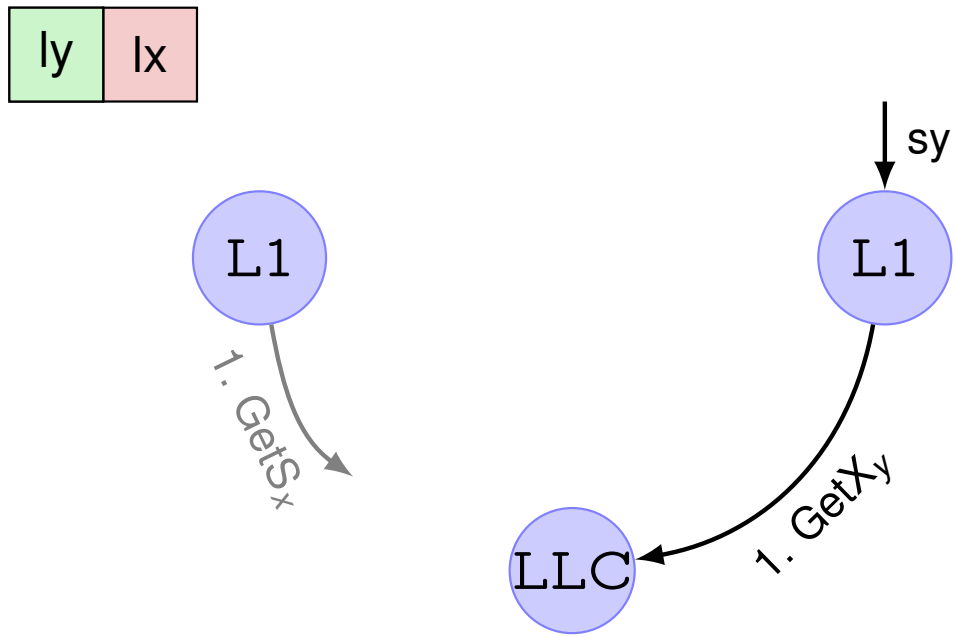


SQUASH OR WAIT?

- Current multicores avoid non-valid results by
  - Interacting with the coherence protocol

```
ly
        sy
        sx
lx

  (1,0)
```

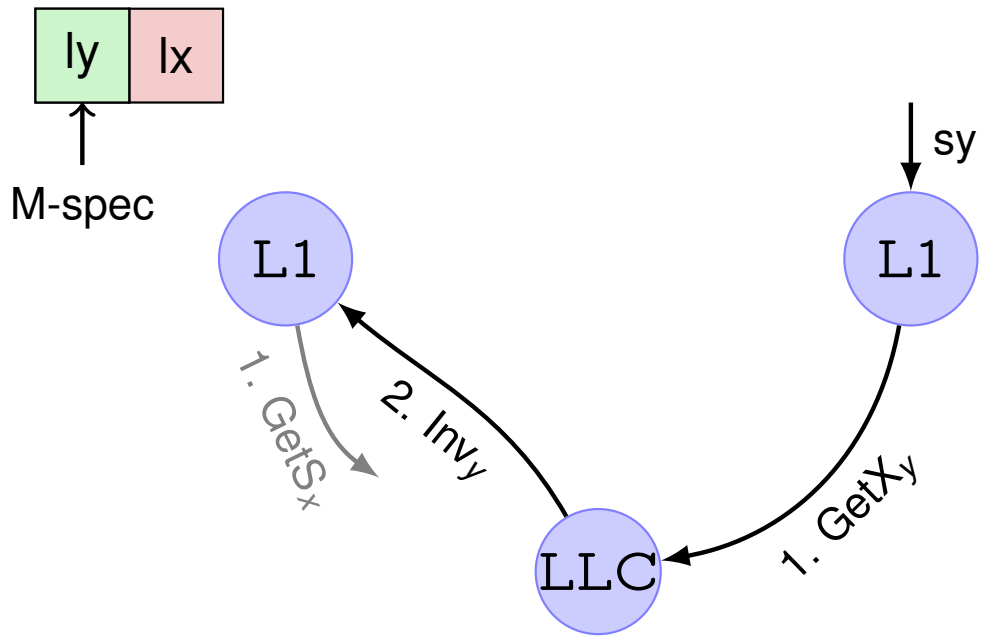


SQUASH OR WAIT?

- Current multicores avoid non-valid results by
  - Interacting with the coherence protocol

```
ly
        sy
        sx
lx

  (1,0)
```

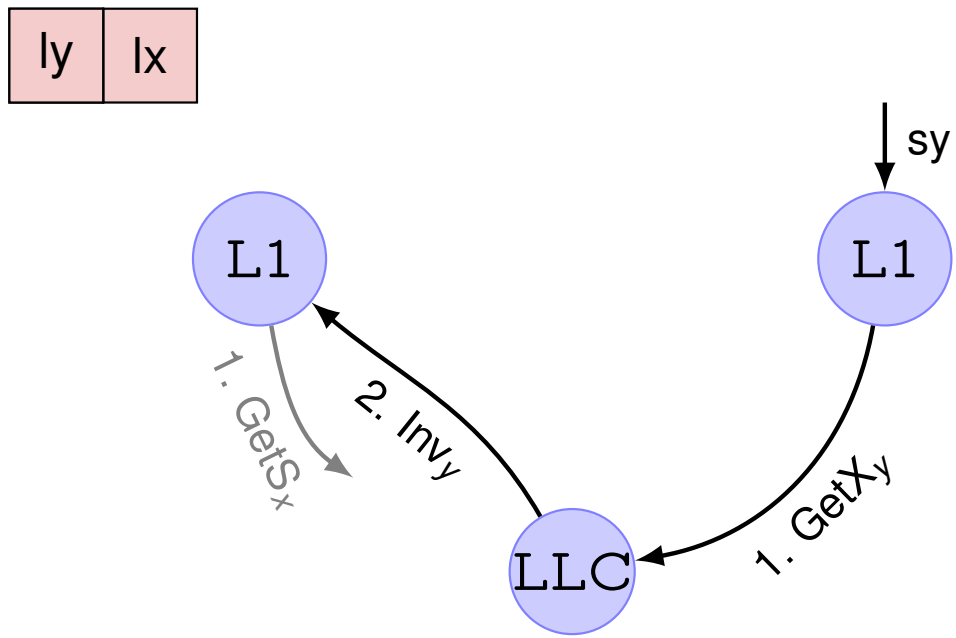


SQUASH OR WAIT?

- Current multicores avoid non-valid results by
  - Interacting with the coherence protocol
  - Squashing and re-executing on remote writes

```
ly
        sy
        sx
lx

  (1,0)
```



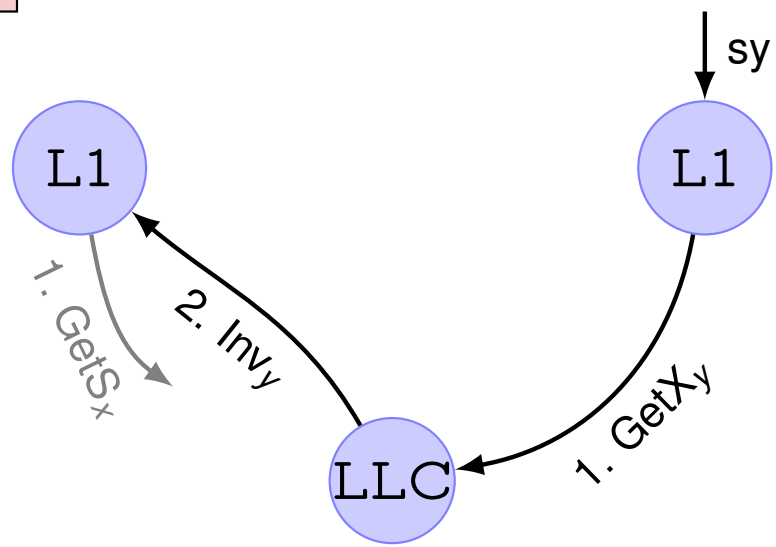
SQUASH OR WAIT?

- Current multicores avoid non-valid results by
  - Interacting with the coherence protocol
  - Squashing and re-executing on remote writes

ly	lx

```
ly                ly
     sy                sy
     sx           lx
lx                     sx
```

(1,0) (0,0)



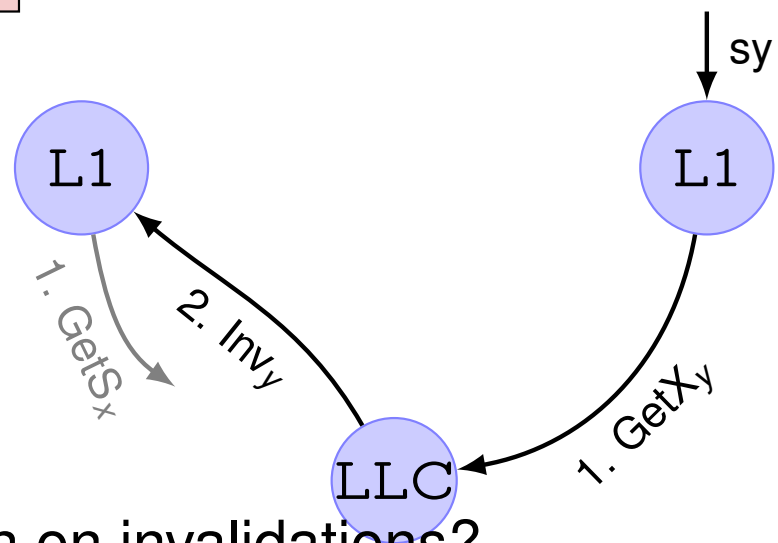
SQUASH OR WAIT?

- Current multicores avoid non-valid results by
  - Interacting with the coherence protocol
  - Squashing and re-executing on remote writes

ly	lx

```
ly                ly
     sy                sy
     sx           lx
lx                     sx

 (1,0)             (0,0)
```



- Do we really need to squash on invalidations?

SQUASH OR WAIT?

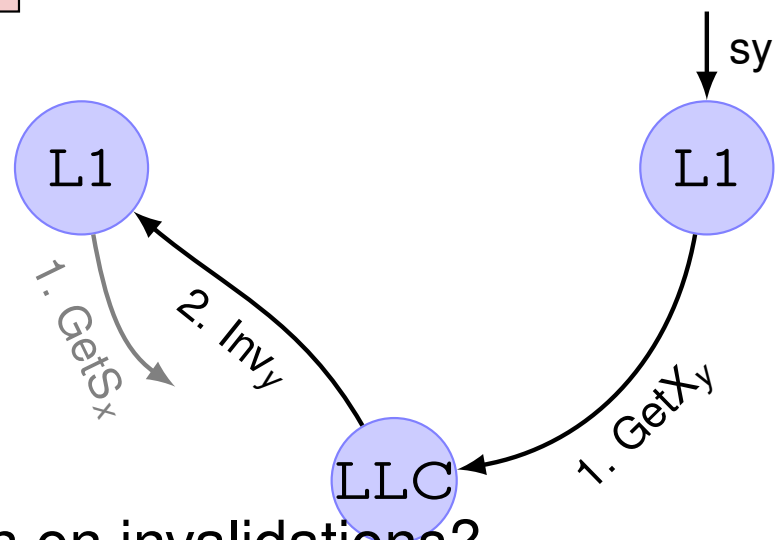
Ros, Carlson, Alipour & Kaxiras [ISCA'17]

- Current multicores avoid non-valid results by
  - Interacting with the coherence protocol
  - Squashing and re-executing on remote writes

ly	lx

```
ly              ly
    sy              sy
    sx          lx
lx                  sx

 (1,0)           (0,0)
```



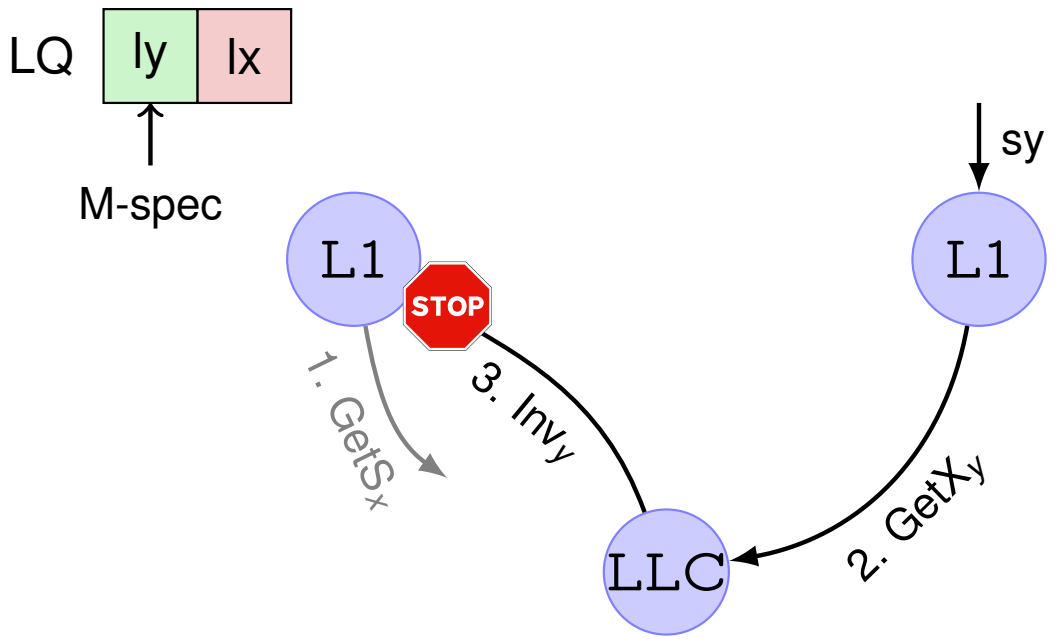
- Do we really need to squash on invalidations?
  - No, we just need to delay `sx`
  - "Non-speculative load-load reordering"

BUT MIND THE DEADLOCKS!

- Blocking writes can cause deadlocks
  - If *x* and *y* are two words within the same cache line

```
ly
        sy
        sx
lx

 (1,0)
```

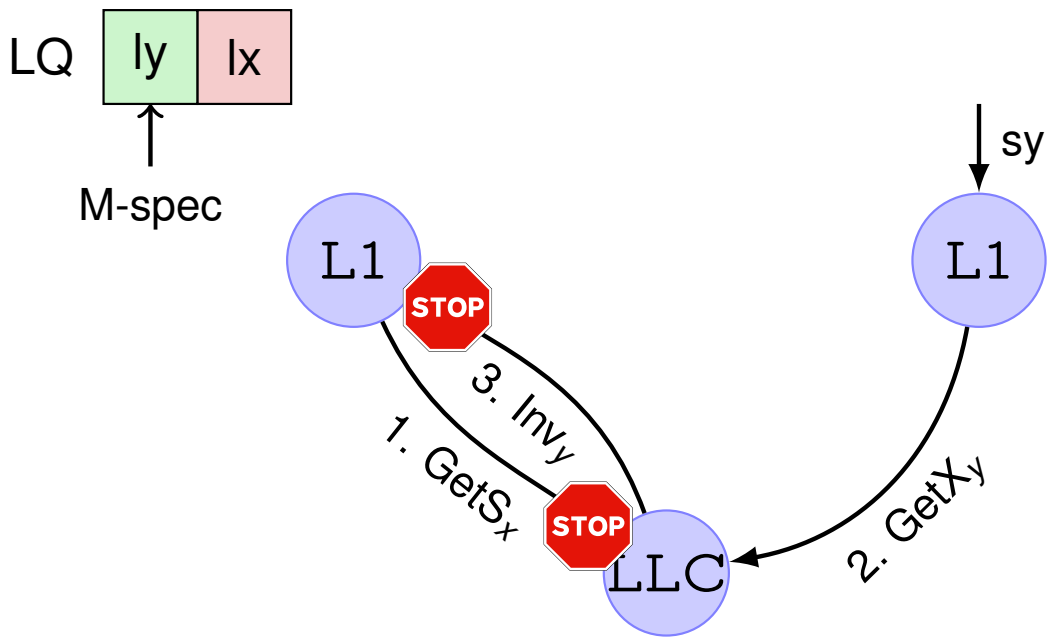


BUT MIND THE DEADLOCKS!

- Blocking writes can cause deadlocks
  - If *x* and *y* are two words within the same cache line

```
ly
        sy
        sx
lx

  (1,0)
```

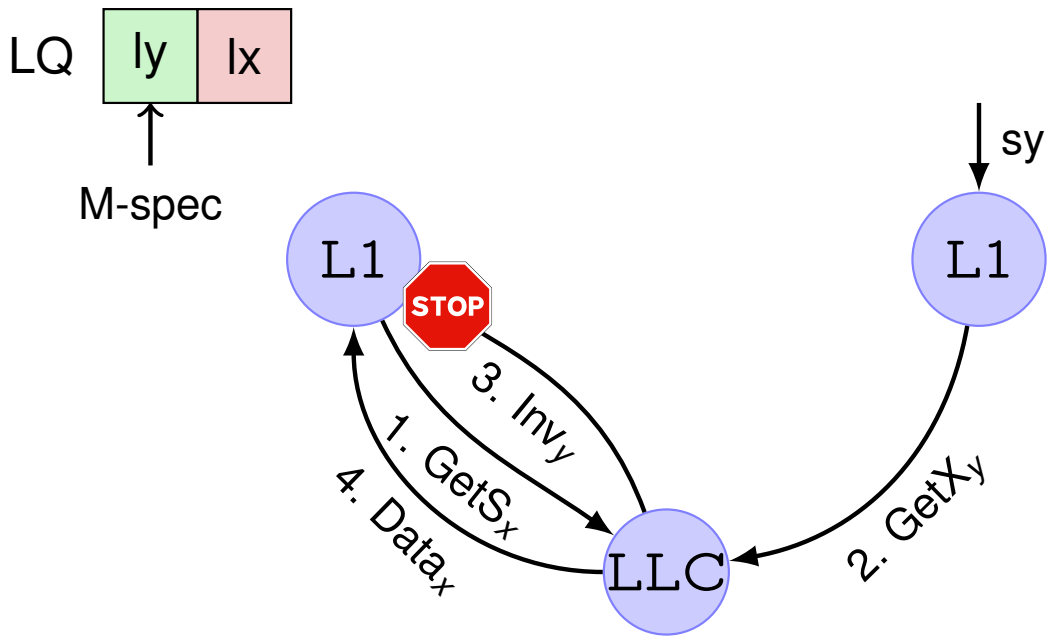


BUT MIND THE DEADLOCKS!

- Blocking writes can cause deadlocks
  - If *x* and *y* are two words within the same cache line
  - SOLUTION: Blocked writes allow reads to be resolved

```
ly
      sy
      sx
lx

 (1,0)
```

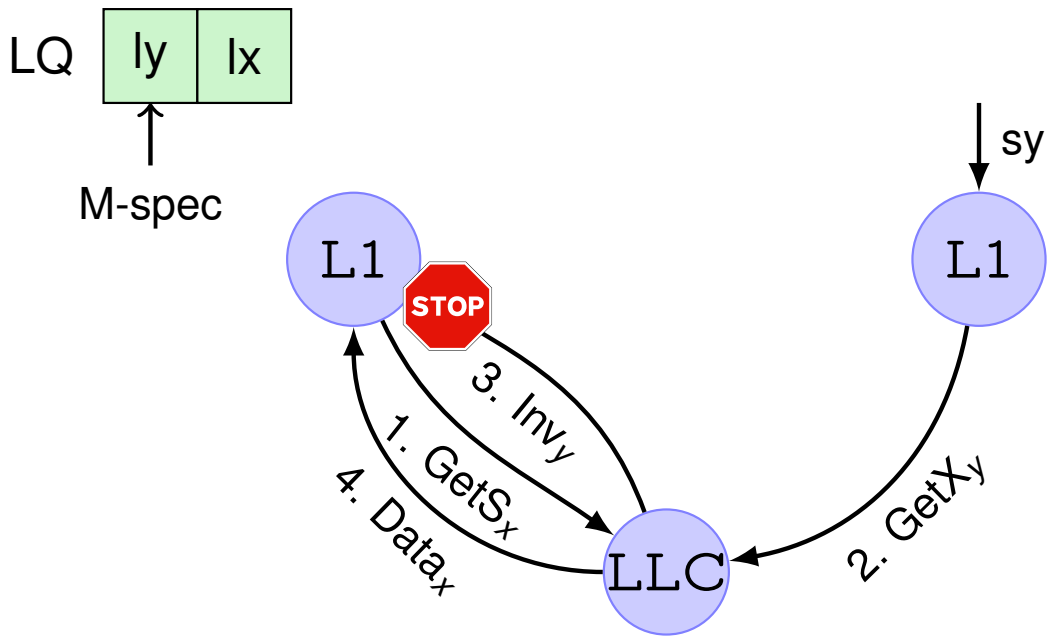


BUT MIND THE DEADLOCKS!

- Blocking writes can cause deadlocks
  - If *x* and *y* are two words within the same cache line
  - SOLUTION: Blocked writes allow reads to be resolved

```
ly
        sy
        sx
lx

  (1,0)
```

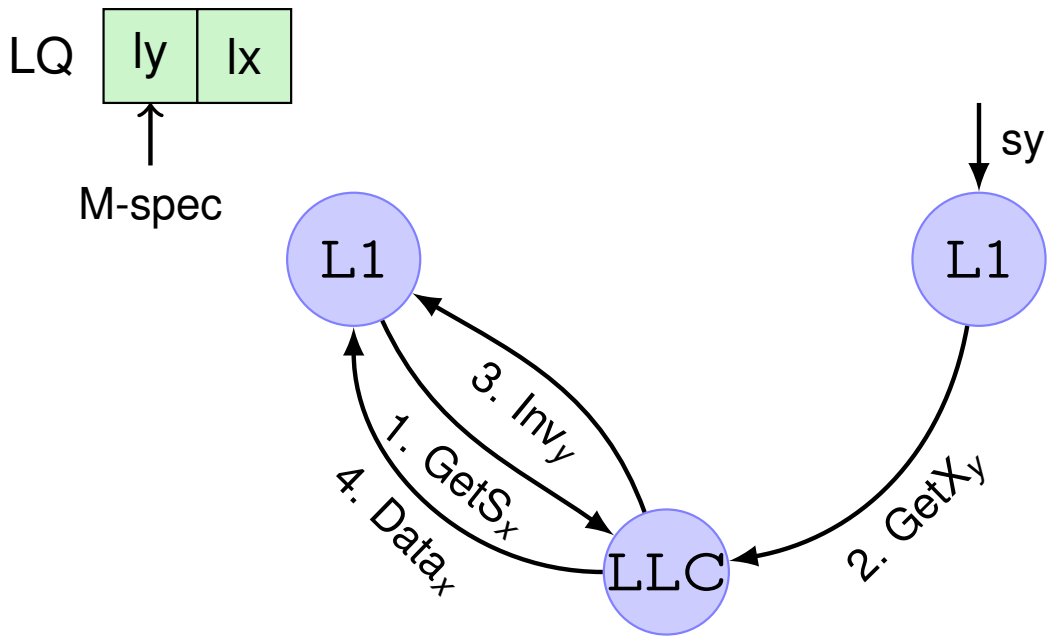


BUT MIND THE DEADLOCKS!

- Blocking writes can cause deadlocks
  - If *x* and *y* are two words within the same cache line
  - SOLUTION: Blocked writes allow reads to be resolved

```
ly
        sy
        sx
lx

  (1,0)
```



ECHO: NON-SPECULATIVE SOLUTIONS

- Disable speculation? $\rightarrow$ Poor performance.

ECHO: NON-SPECULATIVE SOLUTIONS

- Disable speculation? $\rightarrow$ Poor performance.

- NON-SPECULATIVE, HIGH-PERFORMANCE SOLUTION
  What if speculative execution could always succeed?
  It would be transformed into non-speculative execution having the advantages of speculation without:
  - Cost
  - Inefficiencies
  - Security vulnerabilities

ECHO: NON-SPECULATIVE SOLUTIONS

- Disable speculation? $\rightarrow$ Poor performance.

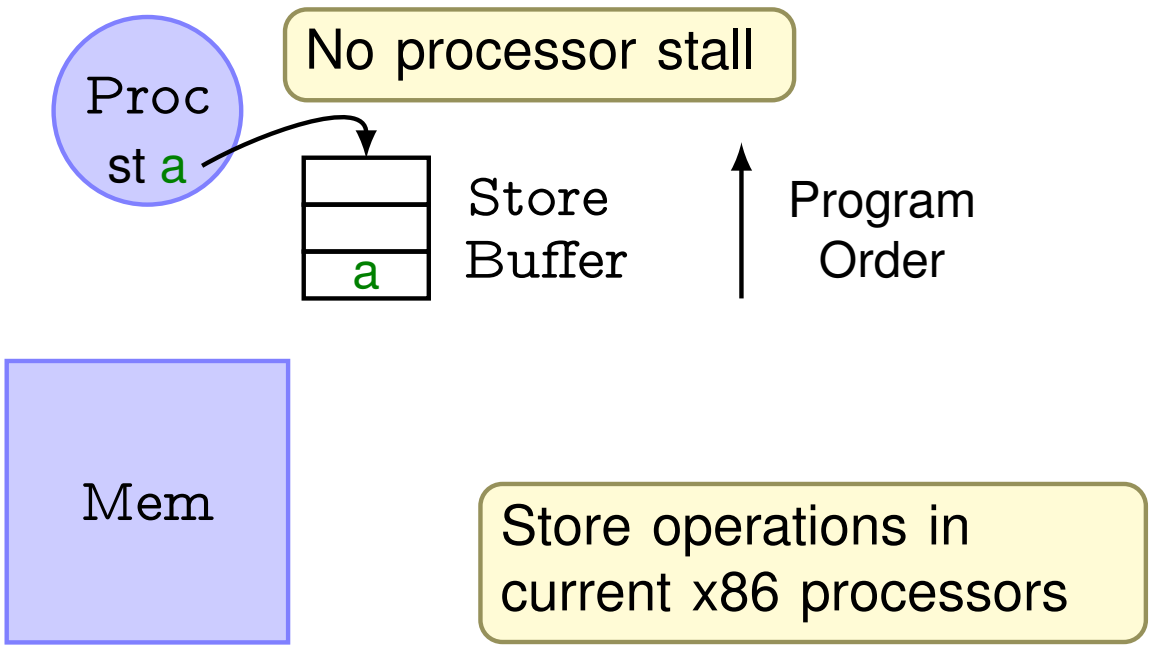
- NON-SPECULATIVE, HIGH-PERFORMANCE SOLUTION
  What if speculative execution could always succeed?
  It would be transformed into non-speculative execution having the advantages of speculation without:
  - Cost
  - Inefficiencies
  - Security vulnerabilities



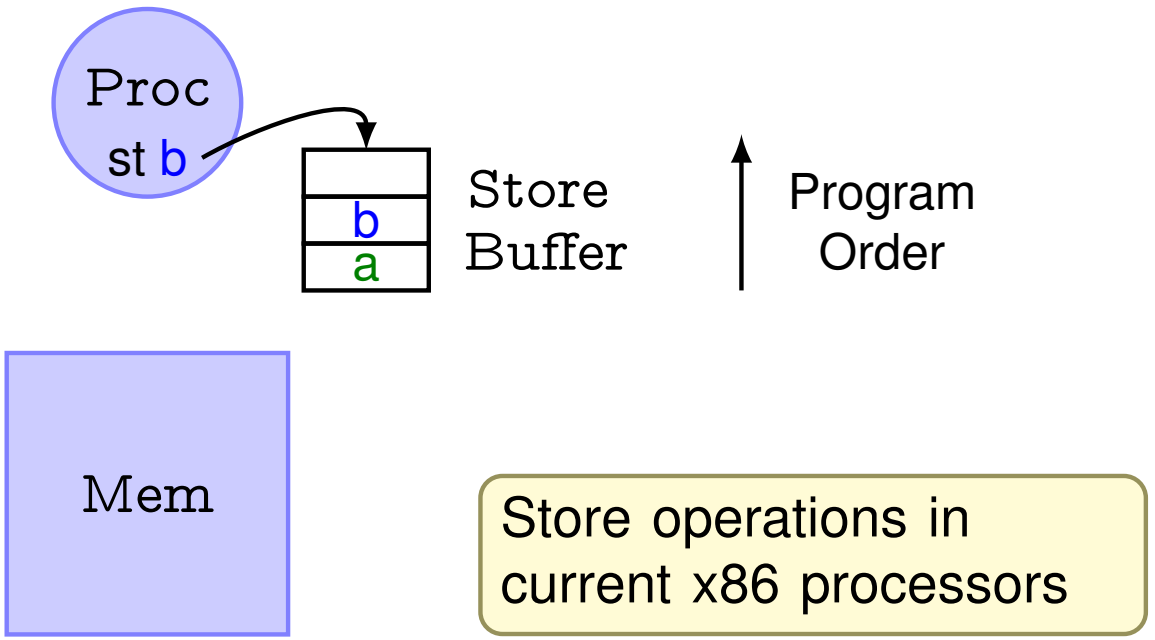
- How?

"The best way to predict your future is to create it."
Abraham Lincoln

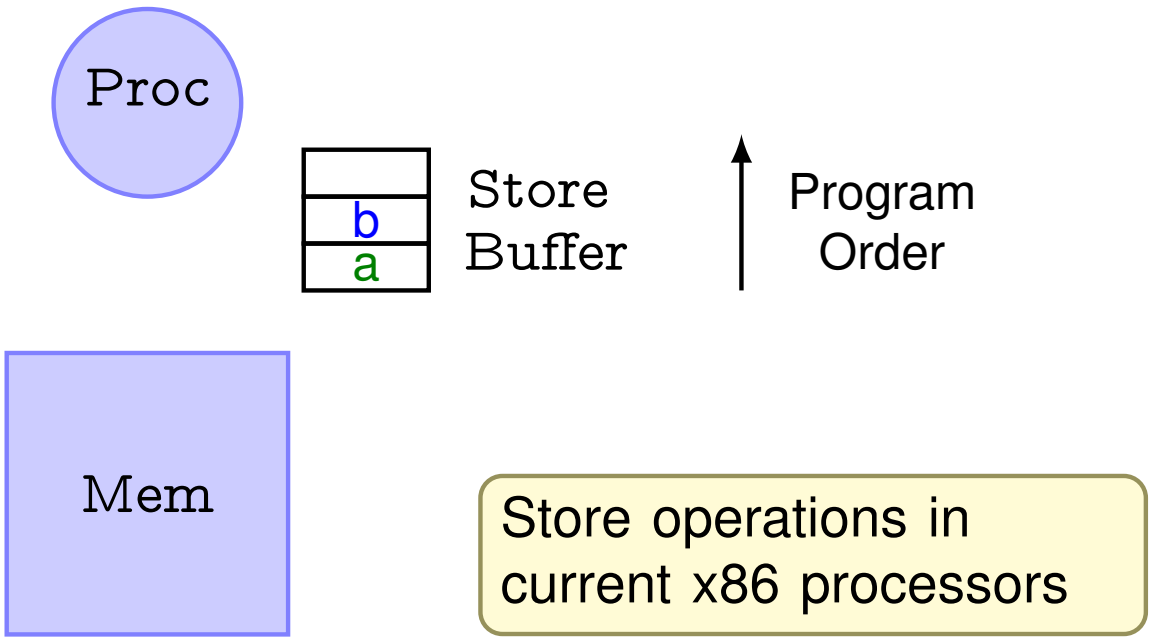
STORE BUFFER AND TOTAL STORE ORDER



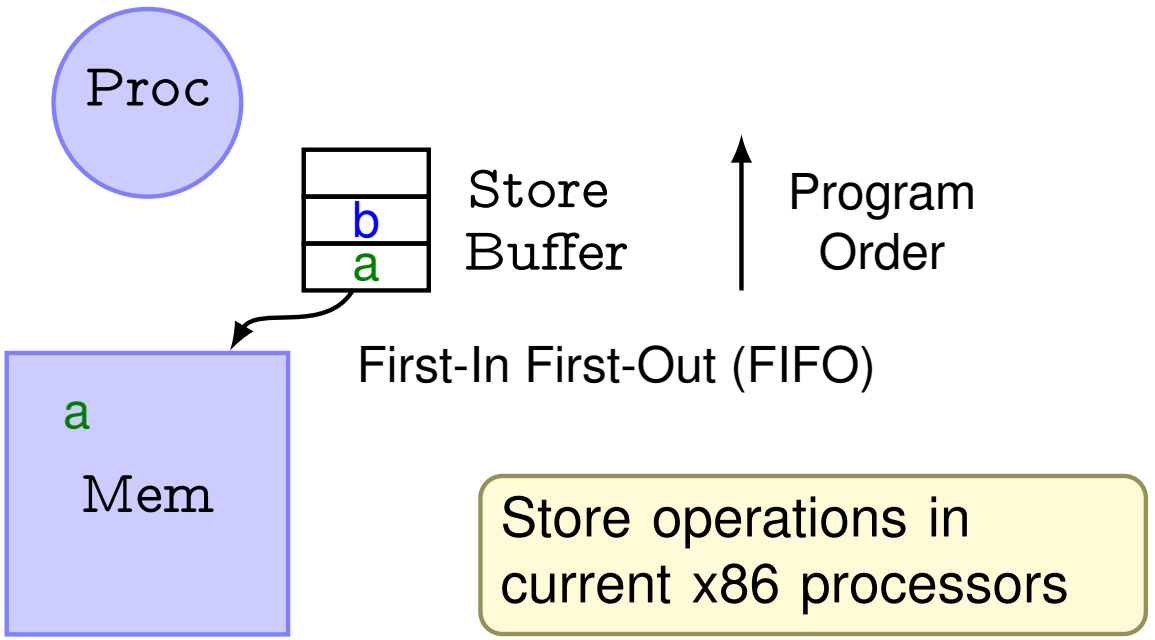
STORE BUFFER AND TOTAL STORE ORDER



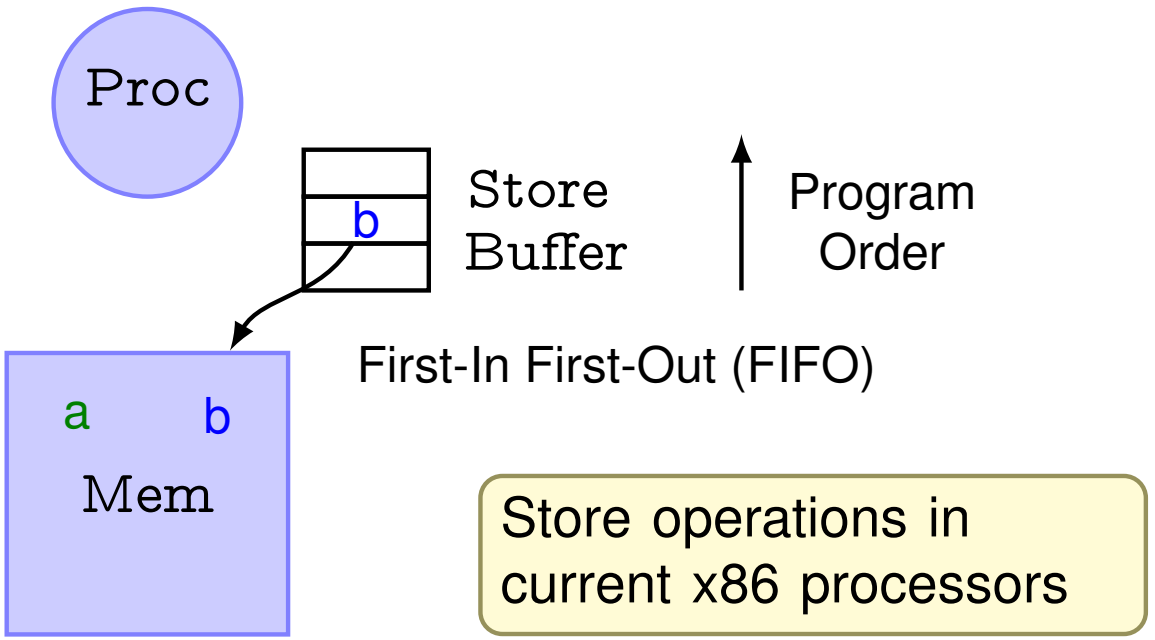
STORE BUFFER AND TOTAL STORE ORDER



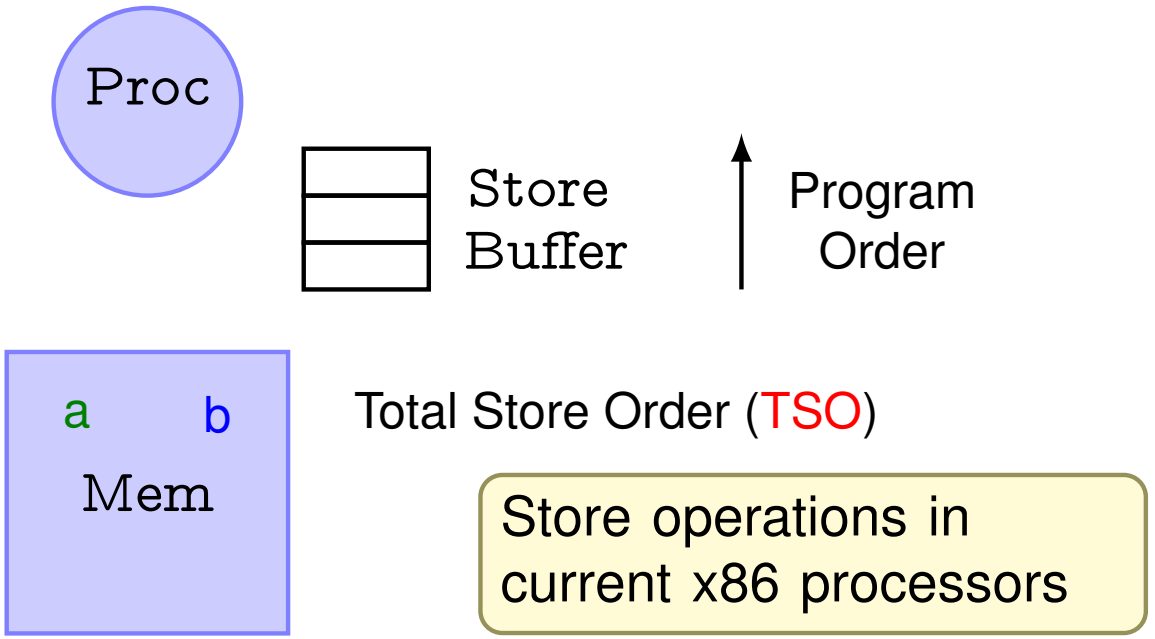
STORE BUFFER AND TOTAL STORE ORDER



STORE BUFFER AND TOTAL STORE ORDER



Store buffer and Total Store Order

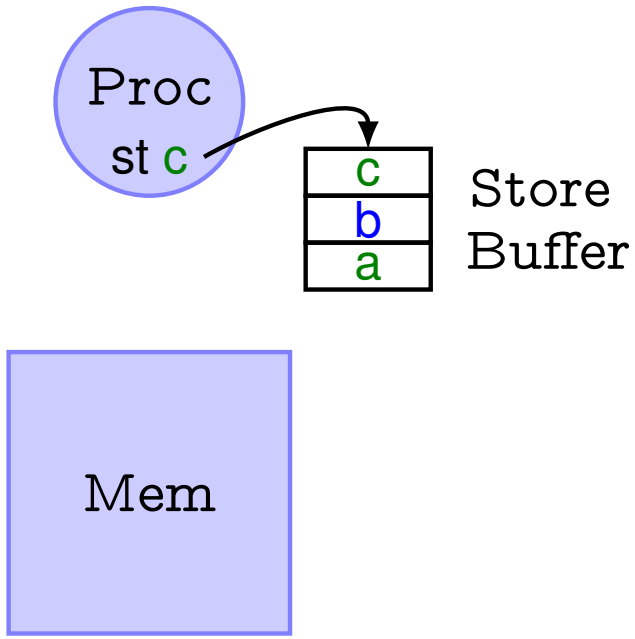


Total Store Order (TSO)

Store operations in current x86 processors

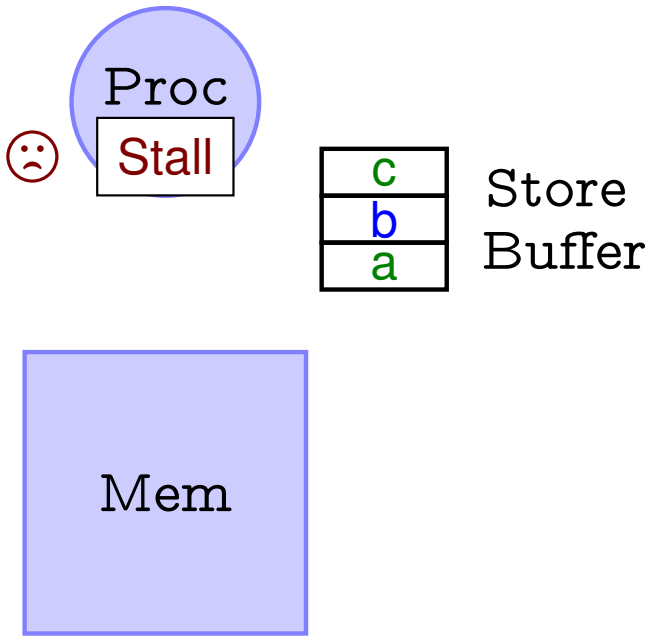
LIMITATIONS AND THE SOLUTION OF COALESCING

Ros & Kaxiras [ISCA'18]



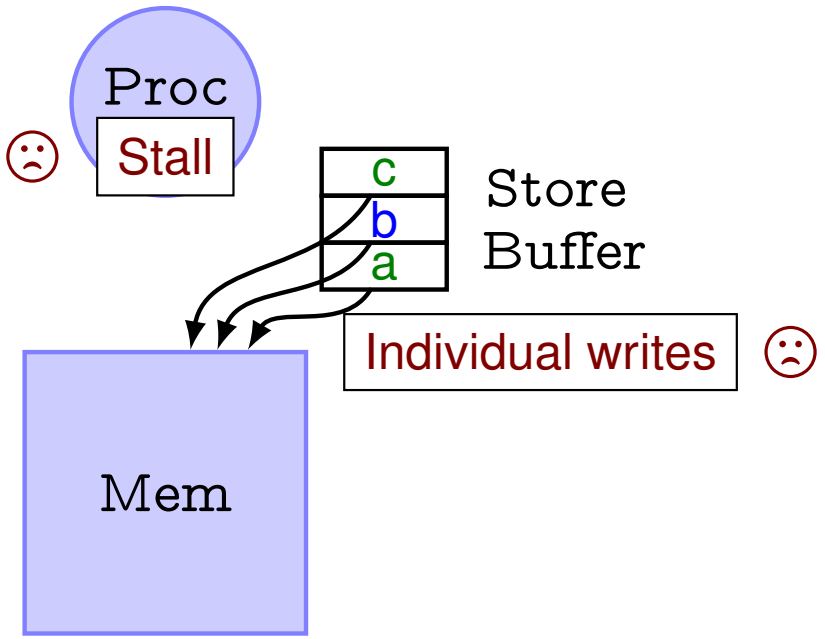
LIMITATIONS AND THE SOLUTION OF COALESCING

Ros & Kaxiras [ISCA'18]



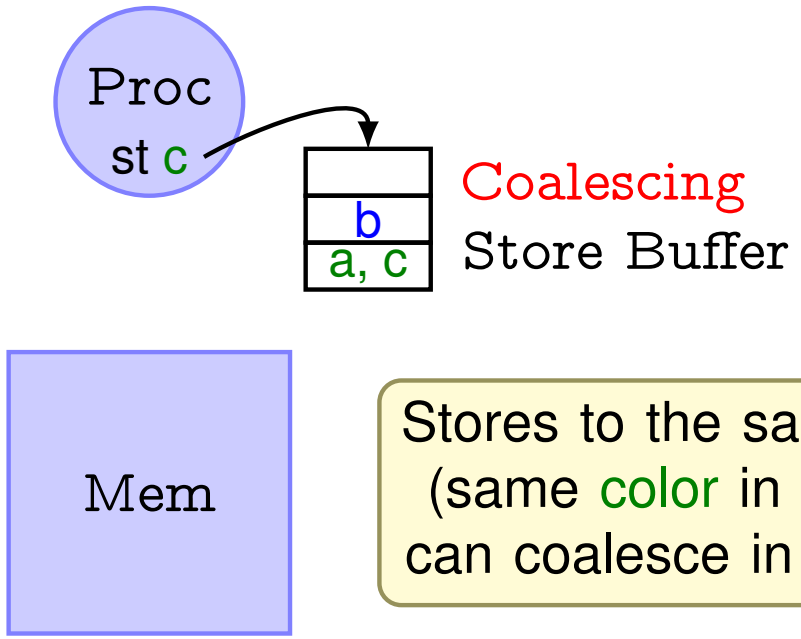
LIMITATIONS AND THE SOLUTION OF COALESCING

Ros & Kaxiras [ISCA'18]



LIMITATIONS AND THE SOLUTION OF COALESCING

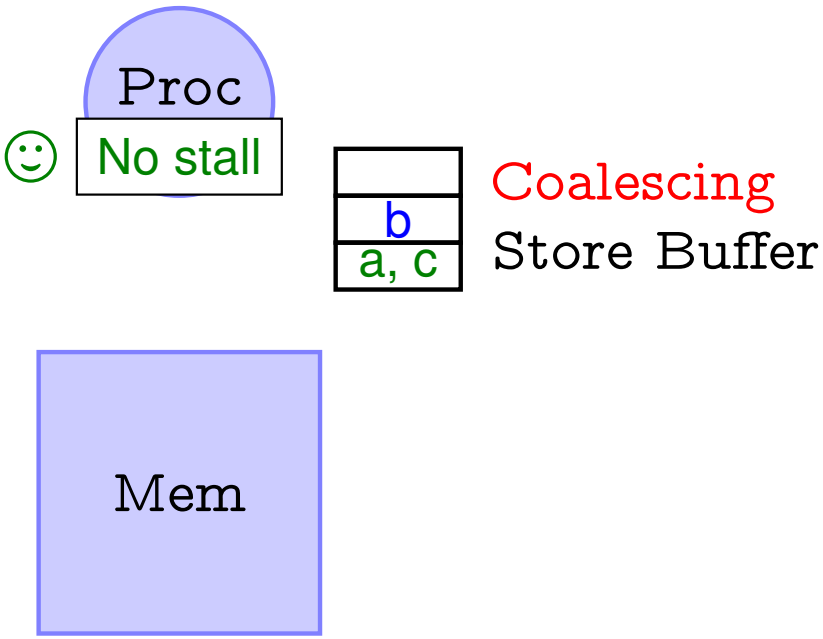
Ros & Kaxiras [ISCA'18]



Stores to the same cache line (same color in the example) can coalesce in a single write

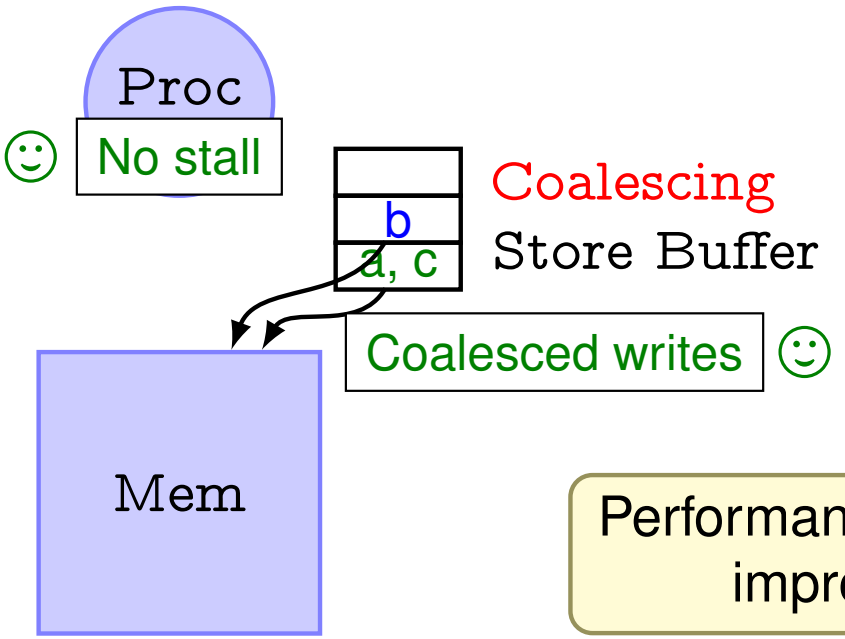
LIMITATIONS AND THE SOLUTION OF COALESCING

Ros & Kaxiras [ISCA'18]



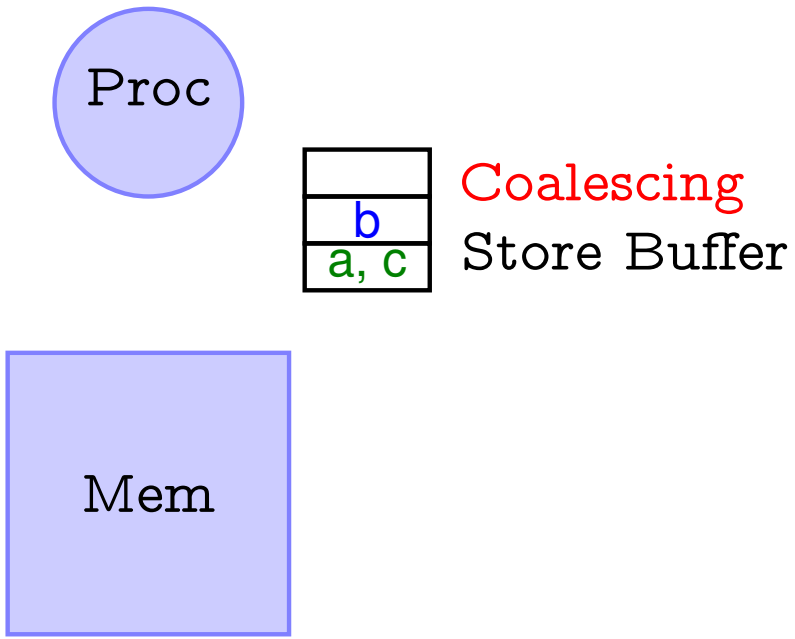
LIMITATIONS AND THE SOLUTION OF COALESCING

Ros & Kaxiras [ISCA'18]

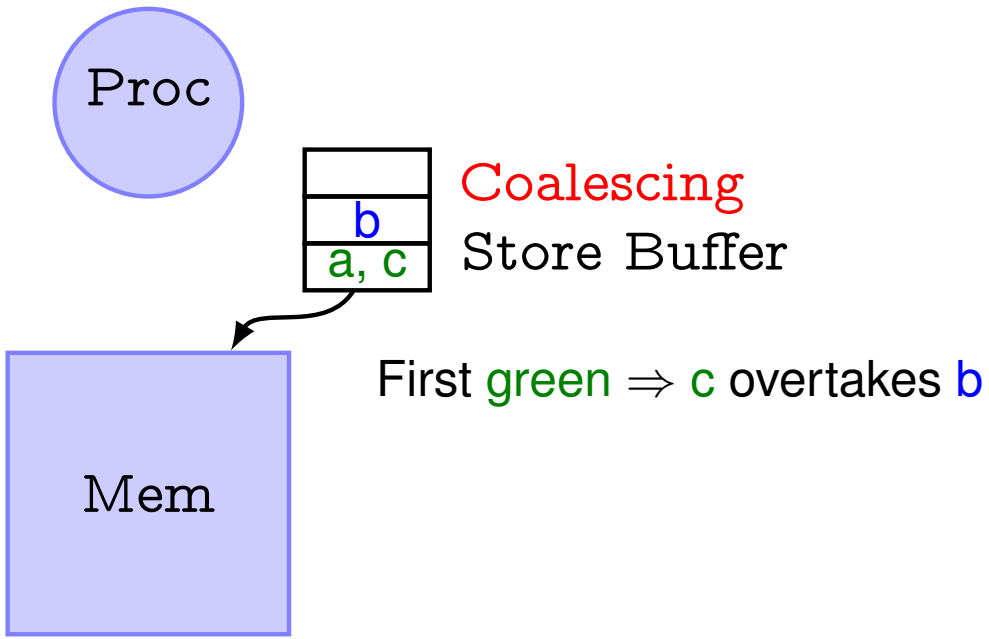


Performance and energy improvements

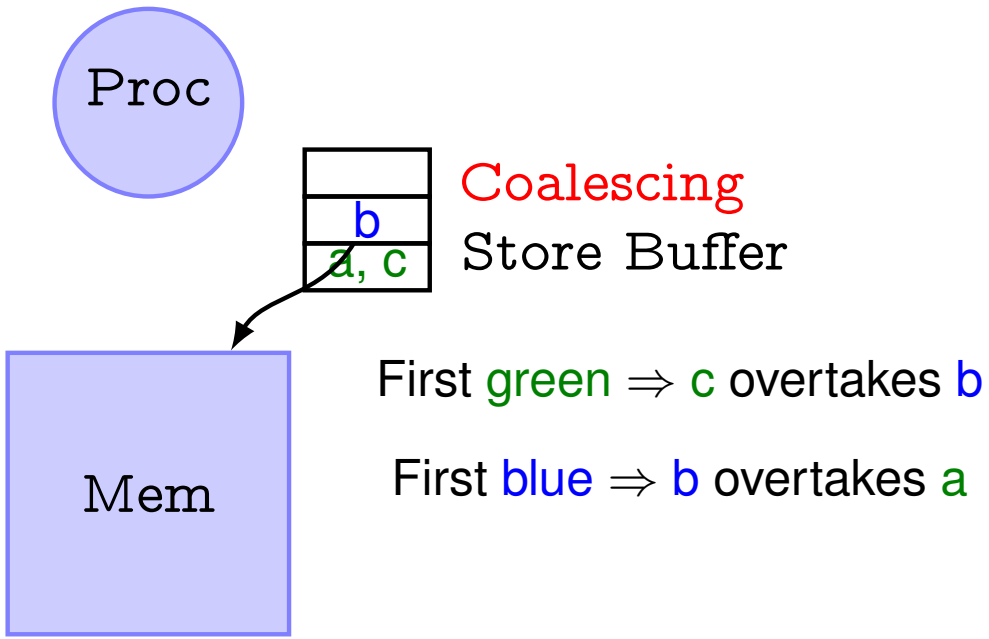
THE PROBLEM OF COALESCING STORES



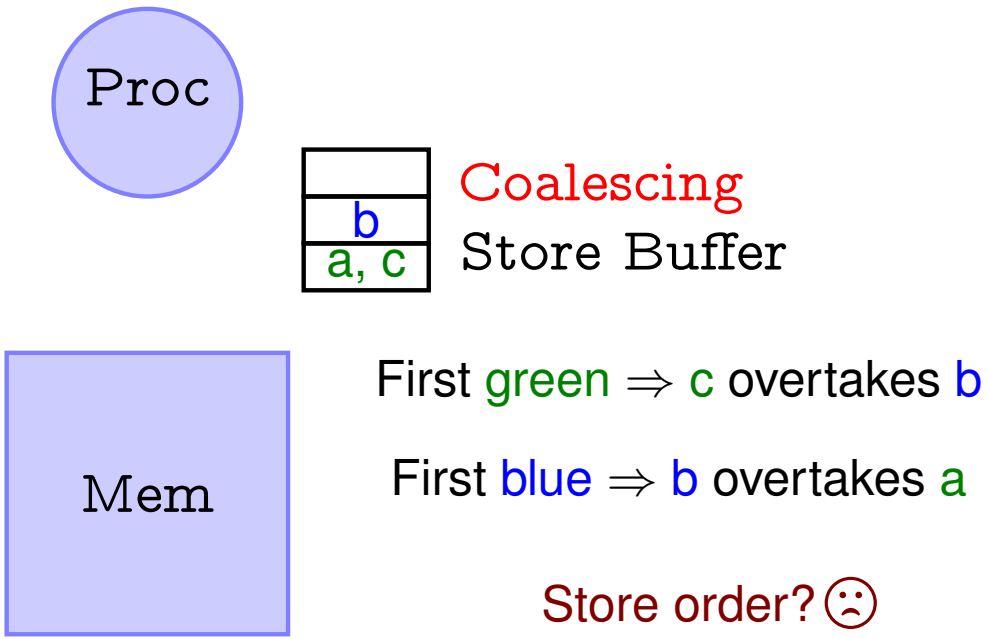
THE PROBLEM OF COALESCING STORES



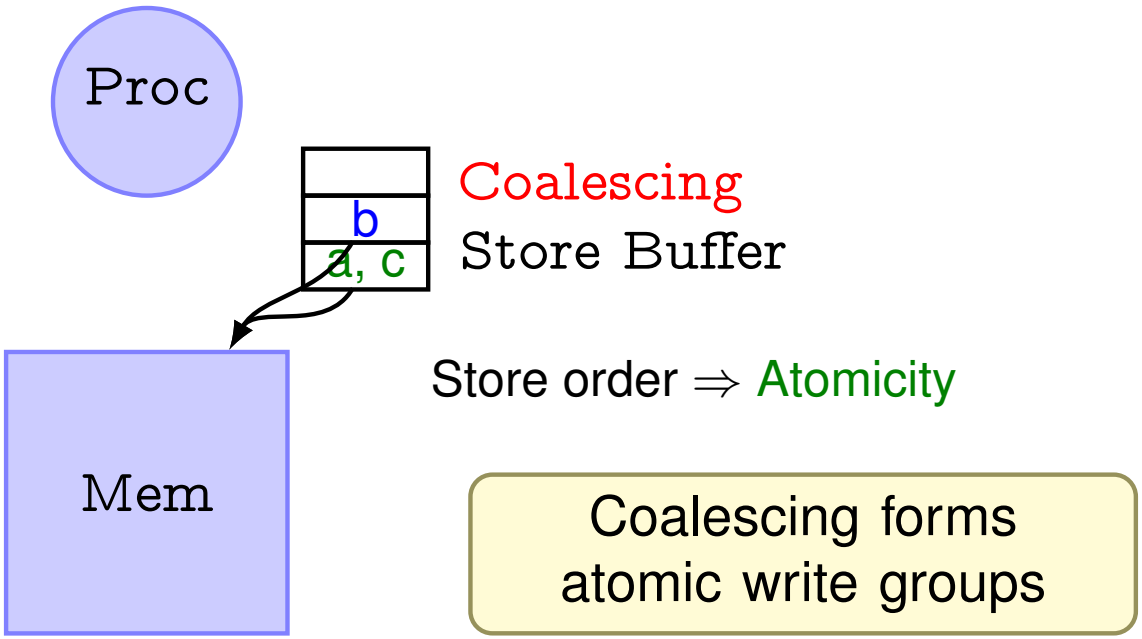
THE PROBLEM OF COALESCING STORES



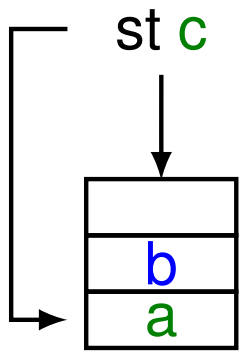
THE PROBLEM OF COALESCING STORES



ATOMICITY: ILLUSION OF STORE ORDER

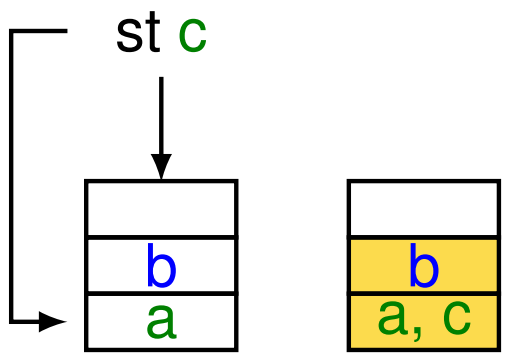


FORMING ATOMIC WRITE GROUPS



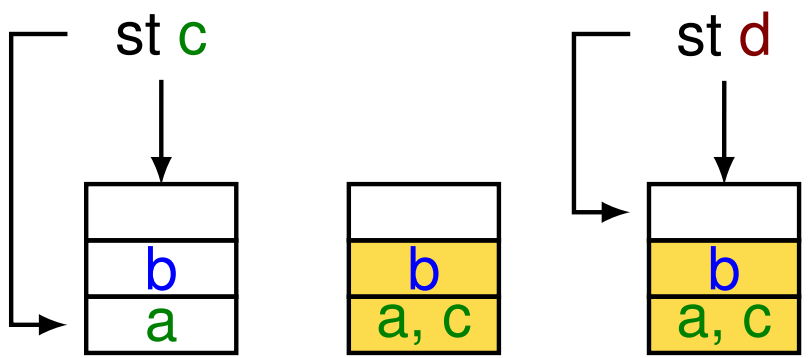
Match younger write to the same cache line

FORMING ATOMIC WRITE GROUPS

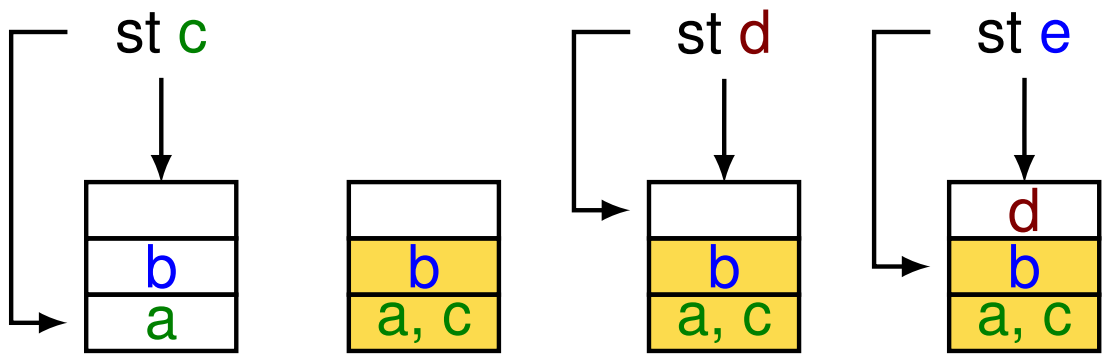


Writes to a, b, and c are indivisible

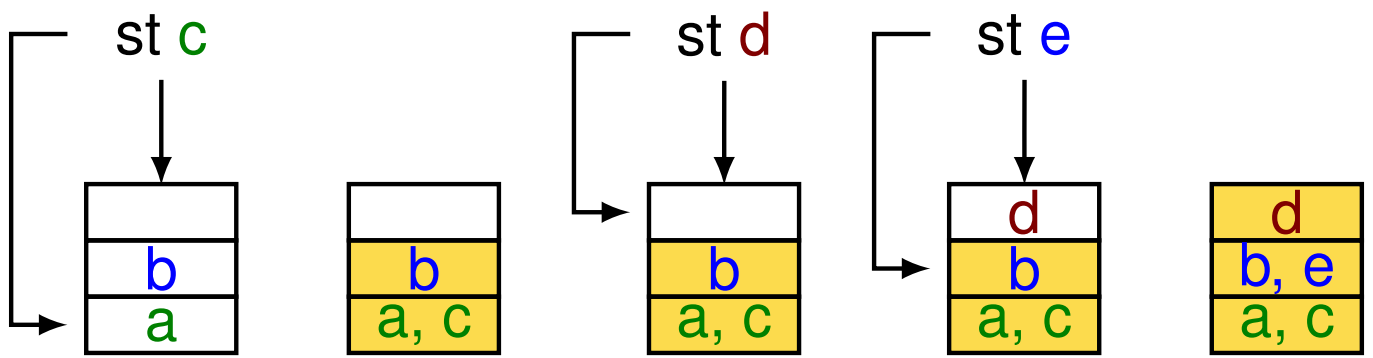
FORMING ATOMIC WRITE GROUPS



FORMING ATOMIC WRITE GROUPS



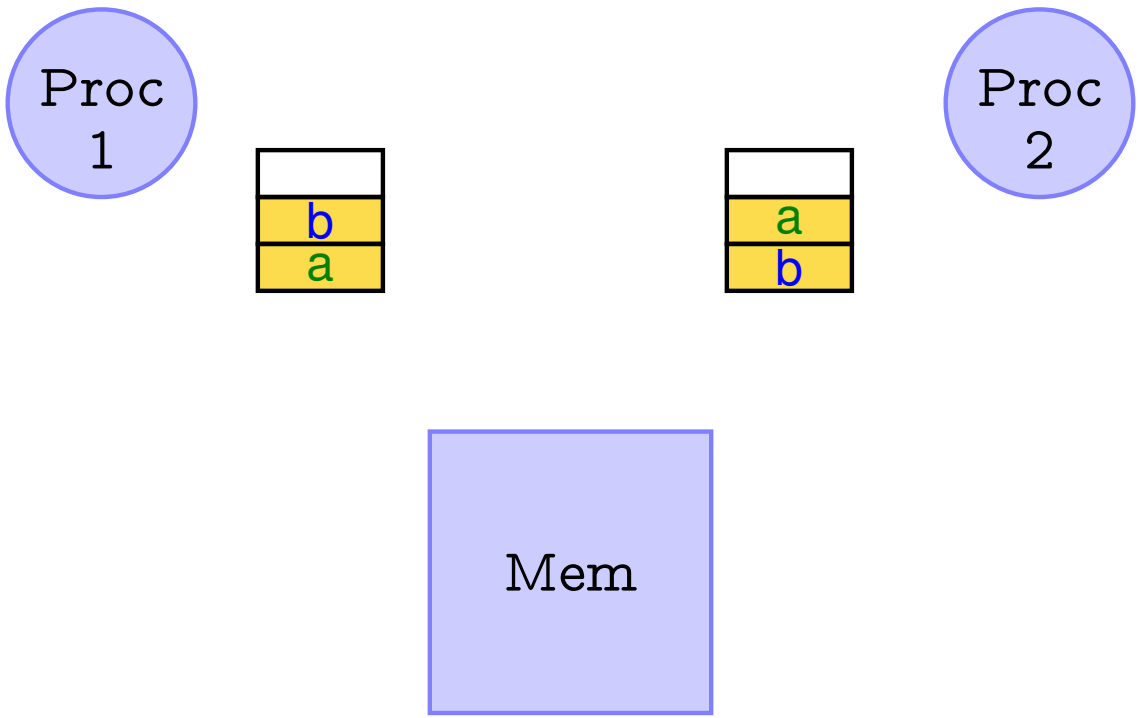
FORMING ATOMIC WRITE GROUPS



Writes to a, b, c, d, and e are indivisible

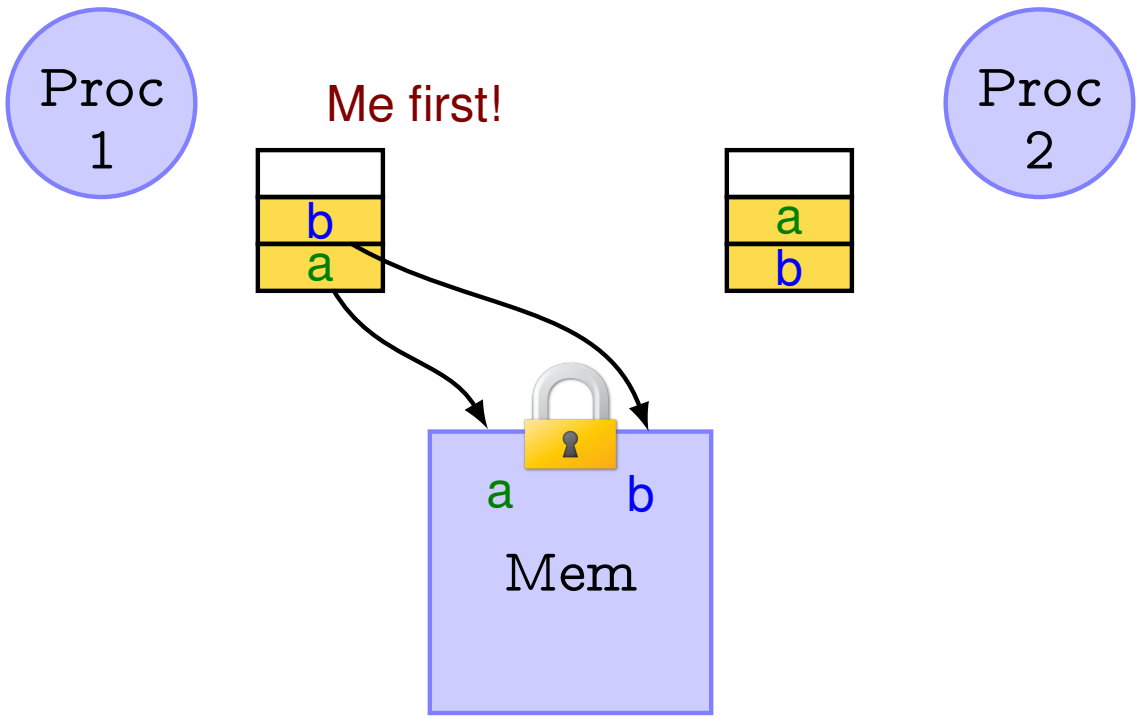
KNOWN WAYS TO PERFORM WRITES ATOMICALY

1. Mutual exclusion (TCC *ISCA'04*, BulkSC *ISCA'07*)
2. Transactional (Ocklahoma *PDTSA'93*, Store-Wait-Free *ISCA'07*)



KNOWN WAYS TO PERFORM WRITES ATOMICALY

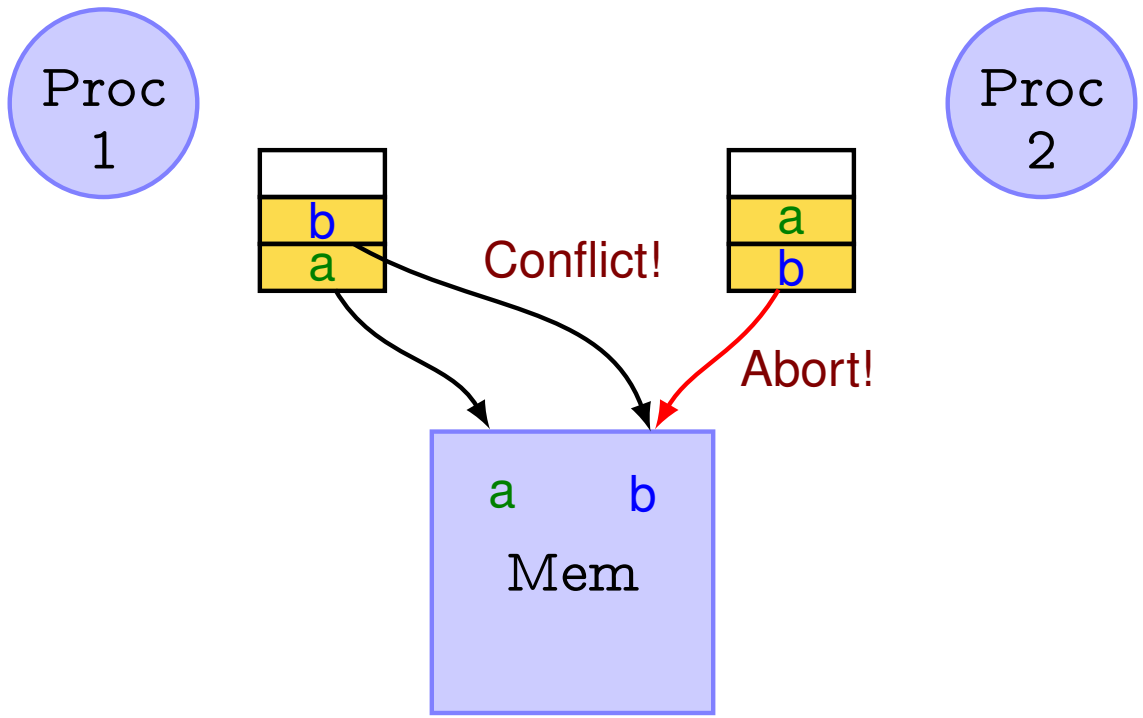
1. Mutual exclusion (TCC *ISCA'04*, BulkSC *ISCA'07*)
2. Transactional (Ocklahoma *PDTSA'93*, Store-Wait-Free *ISCA'07*)



Centralized and non-scalable solution

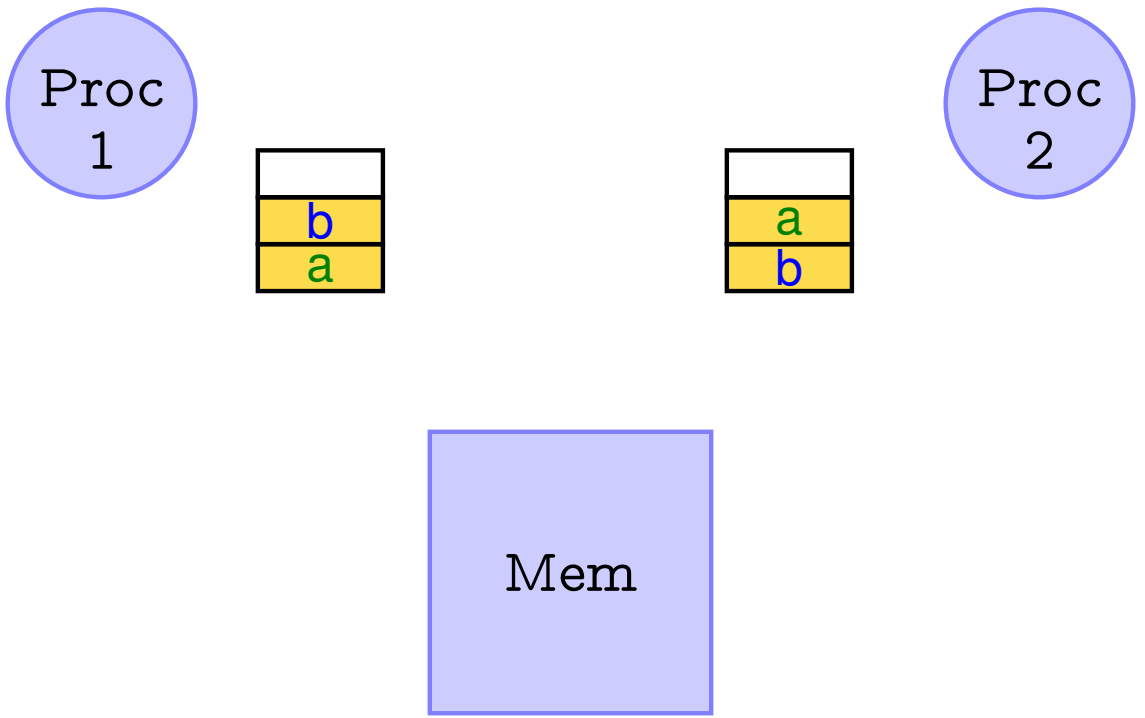
KNOWN WAYS TO PERFORM WRITES ATOMICALY

1. Mutual exclusion (TCC *ISCA'04*, BulkSC *ISCA'07*)
2. Transactional (Ocklahoma *PDTSA'93*, Store-Wait-Free *ISCA'07*)



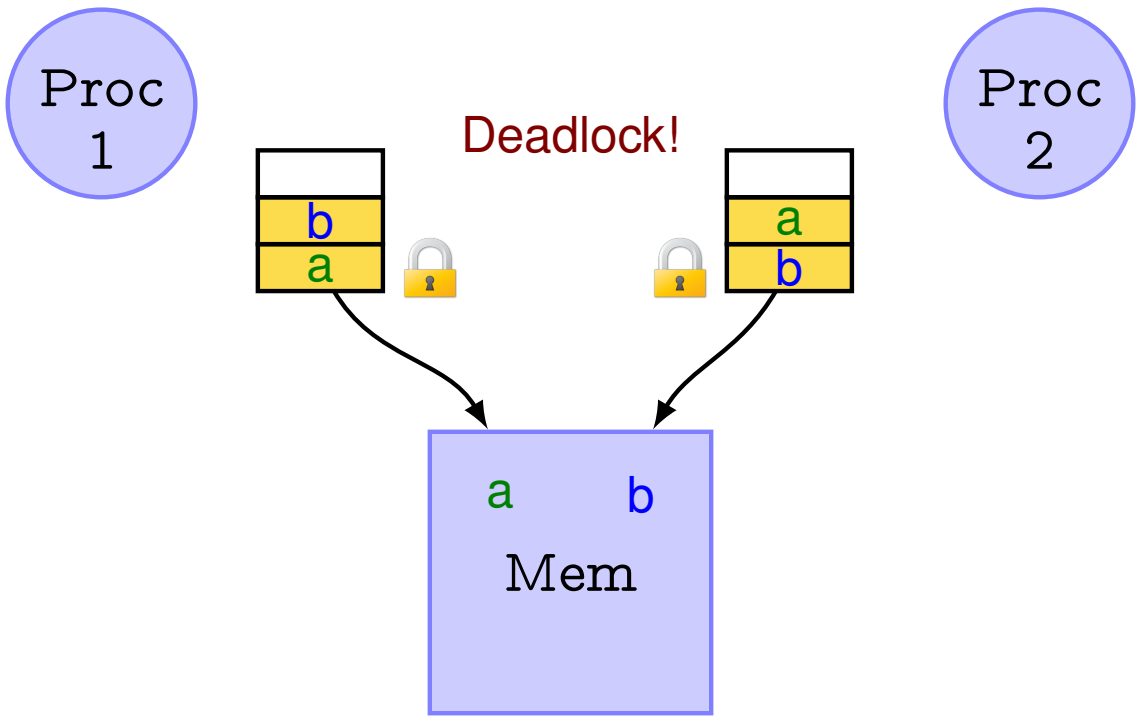
- Speculation: rollback on conflict
- Canceling memory writes is a costly operation

A NON-SPECULATIVE PERSPECTIVE



- Writing atomically a number of cache lines is similar to the problem of acquiring a number of locks in parallel programming

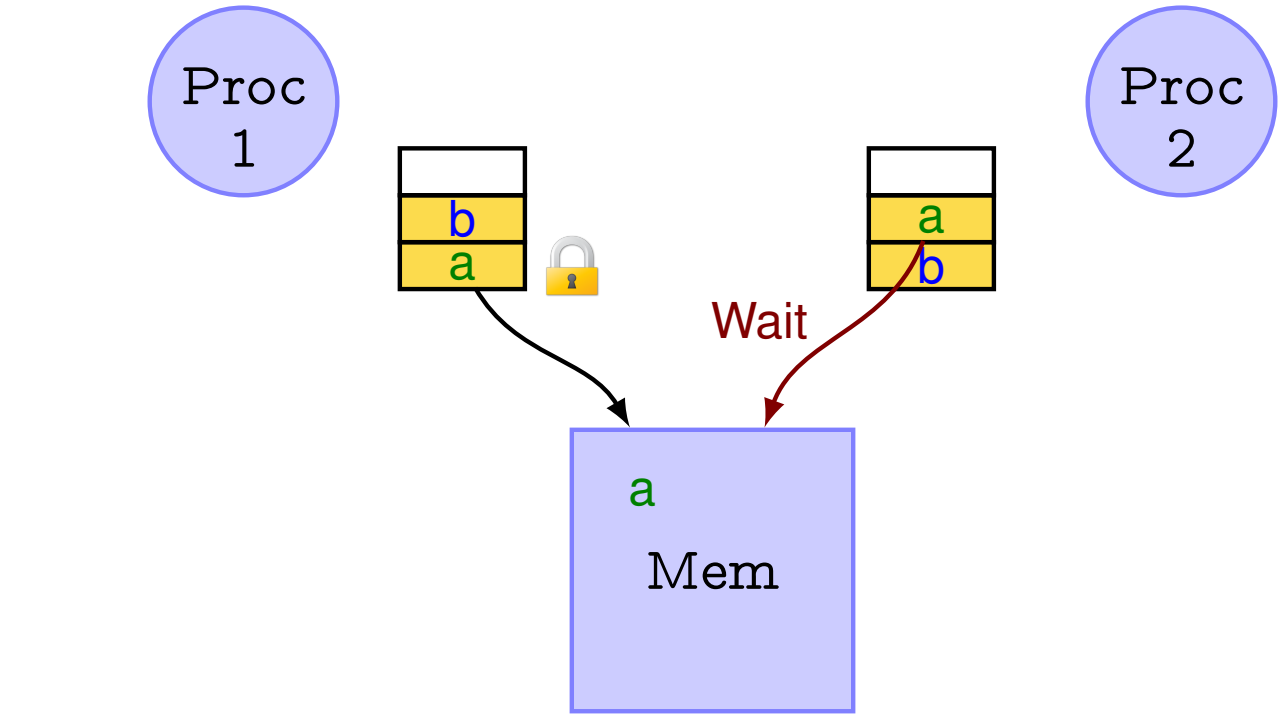
A NON-SPECULATIVE PERSPECTIVE



- Writing atomically a number of cache lines is similar to the problem of acquiring a number of locks in parallel programming
- Deadlock, if locks are taken in opposite order

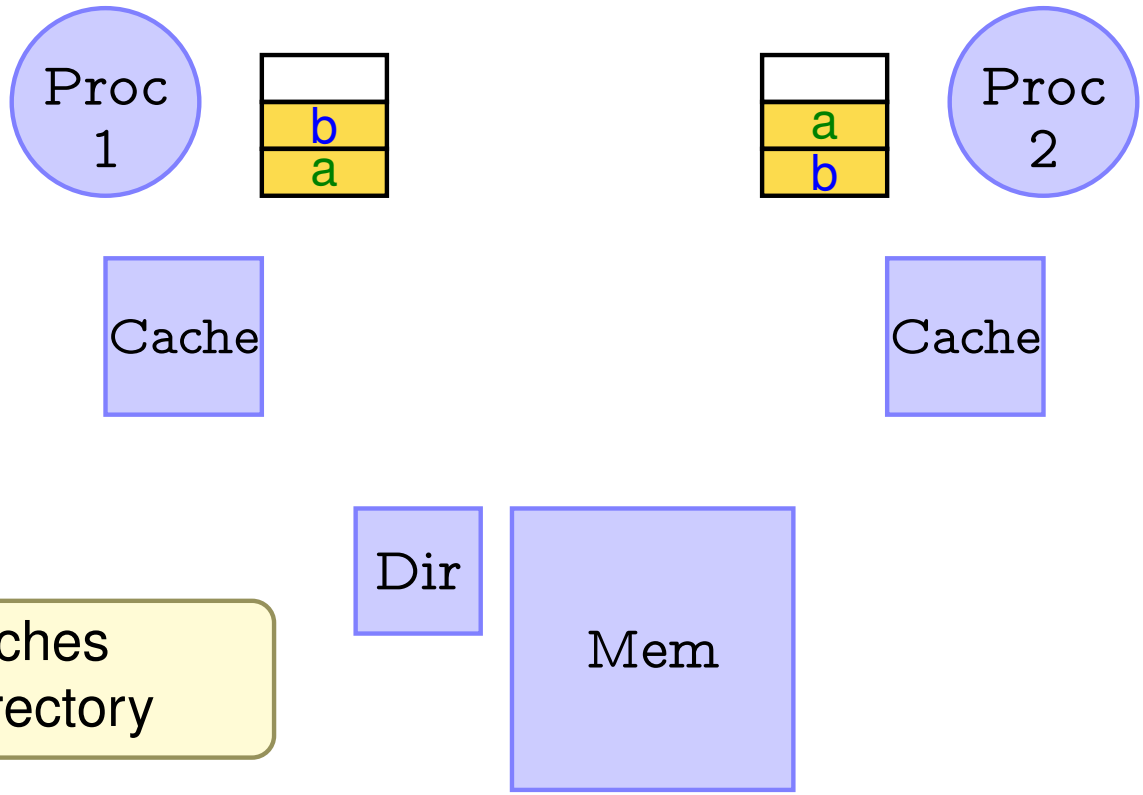
A NON-SPECULATIVE PERSPECTIVE

Perform writes following a global order
Deadlock-free considering unlimited resources[2]



[2] Dijkstra, “Hierarchical ordering of sequential processes”

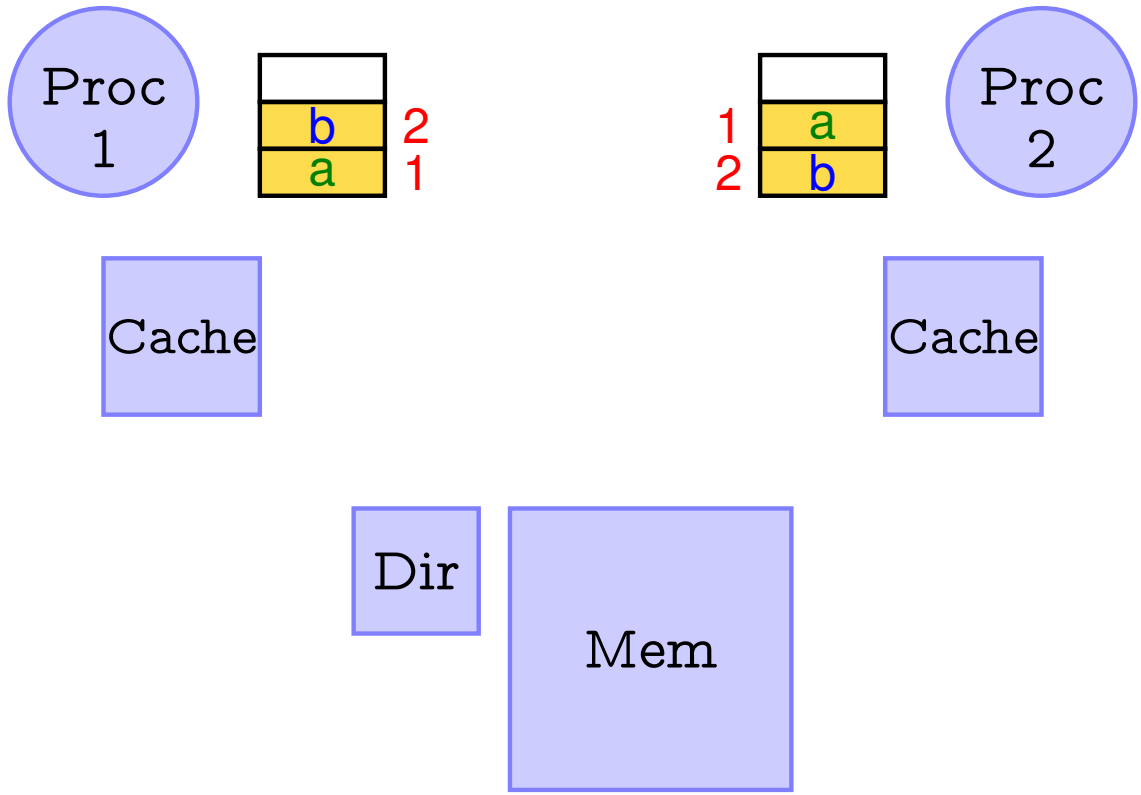
LEXICOGRAPHICAL ORDER



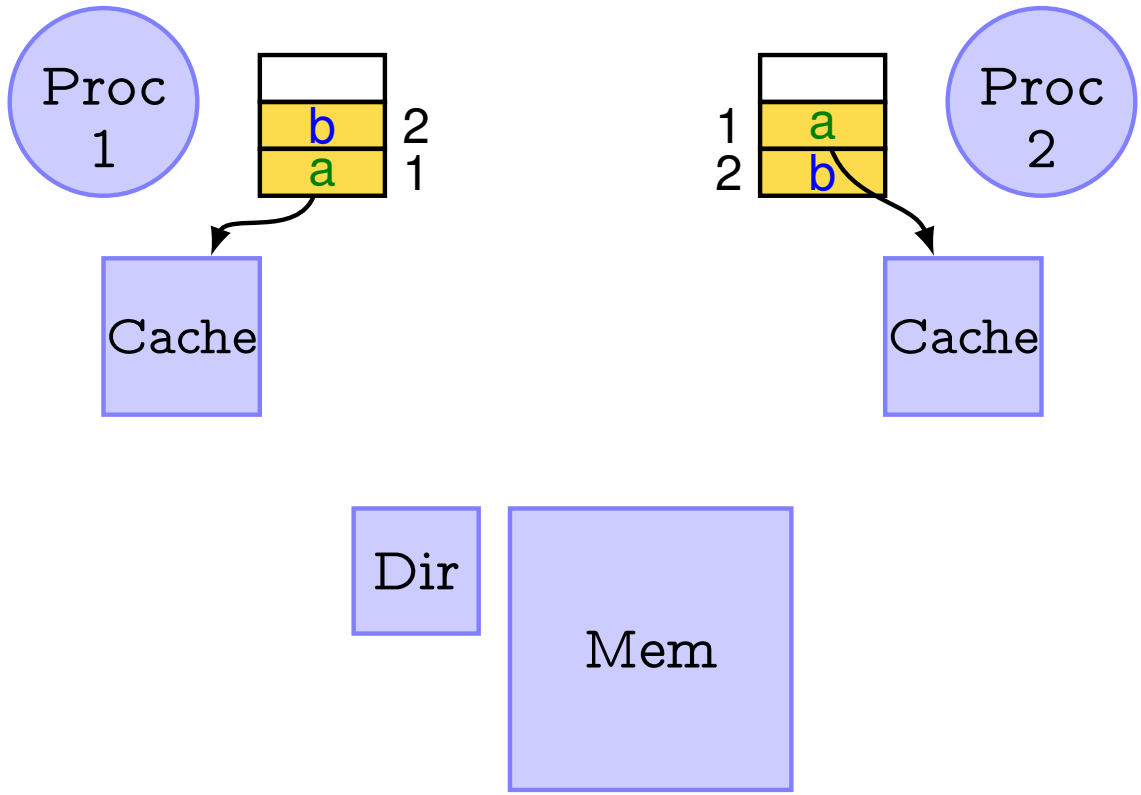
- Private caches
- Shared directory

LEXICOGRAPHICAL ORDER

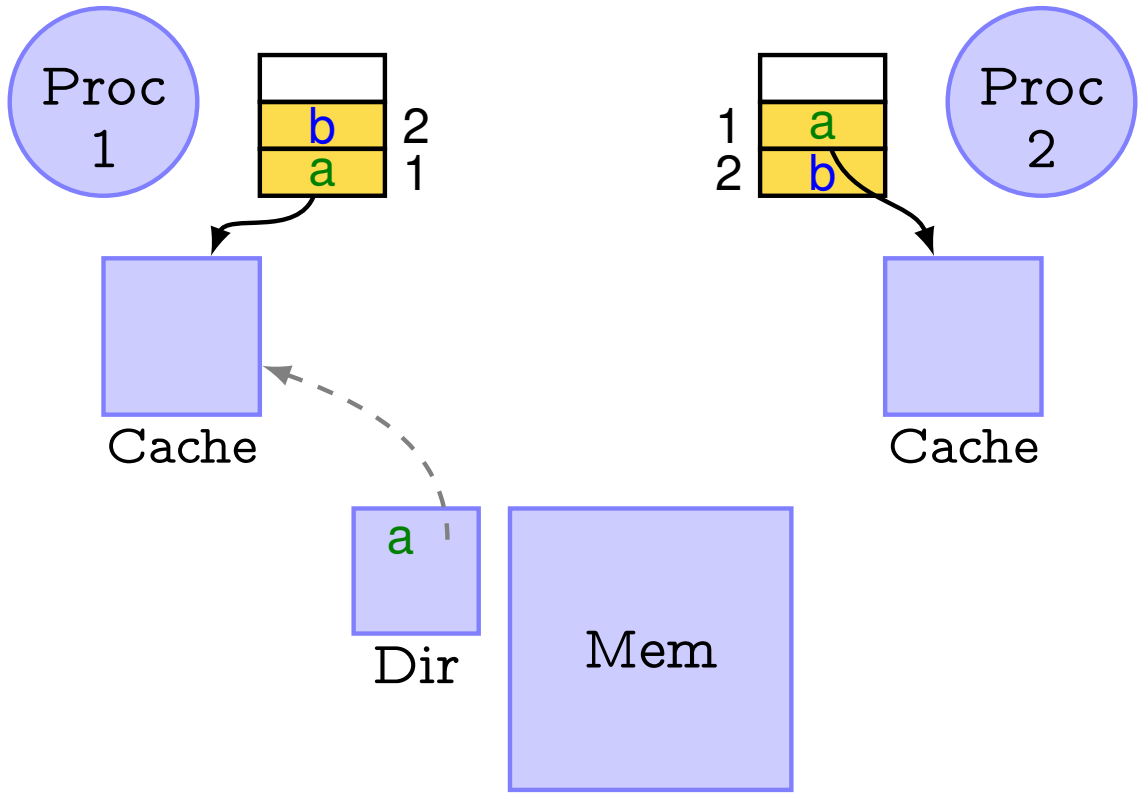
Write in lexicographical (Lex) order $\Rightarrow$ physical address



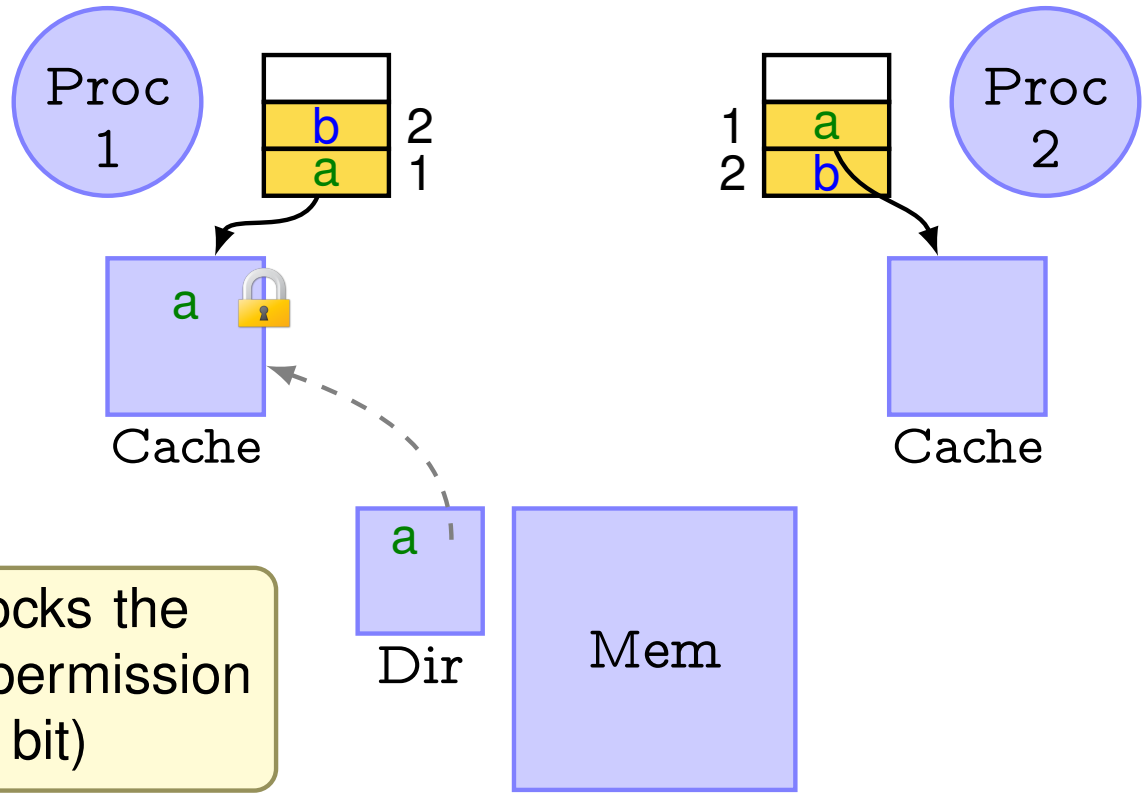
LEXICOGRAPHICAL ORDER



LEXICOGRAPHICAL ORDER



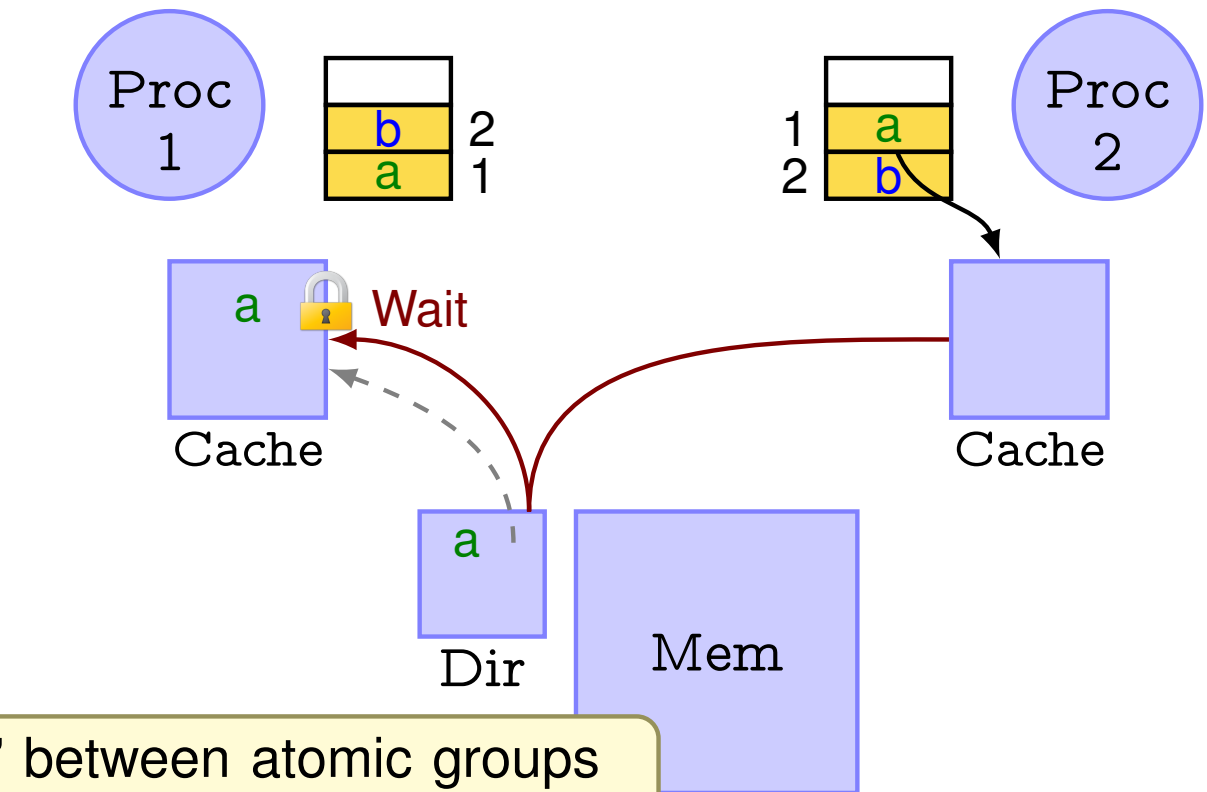
LEXICOGRAPHICAL ORDER



A write locks the cache line permission (lock bit)

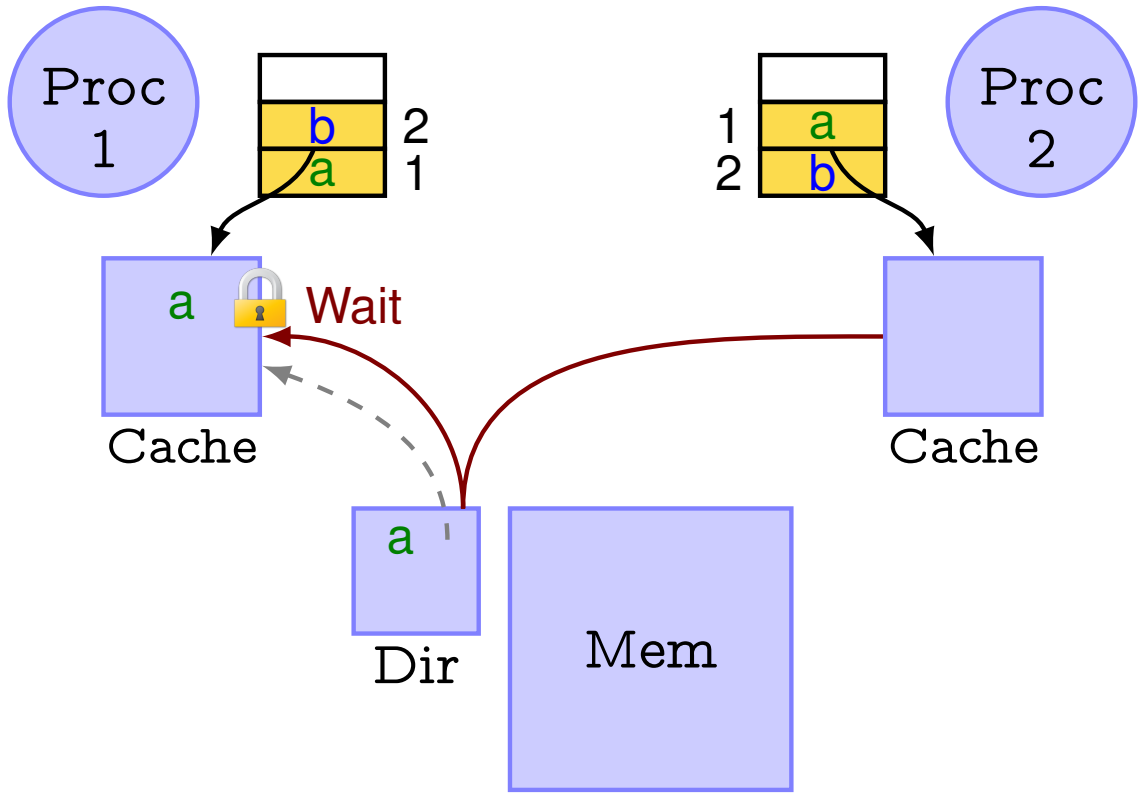
LEXICOGRAPHICAL ORDER

Group writes have been ordered $\Rightarrow$ `Proc 1` first

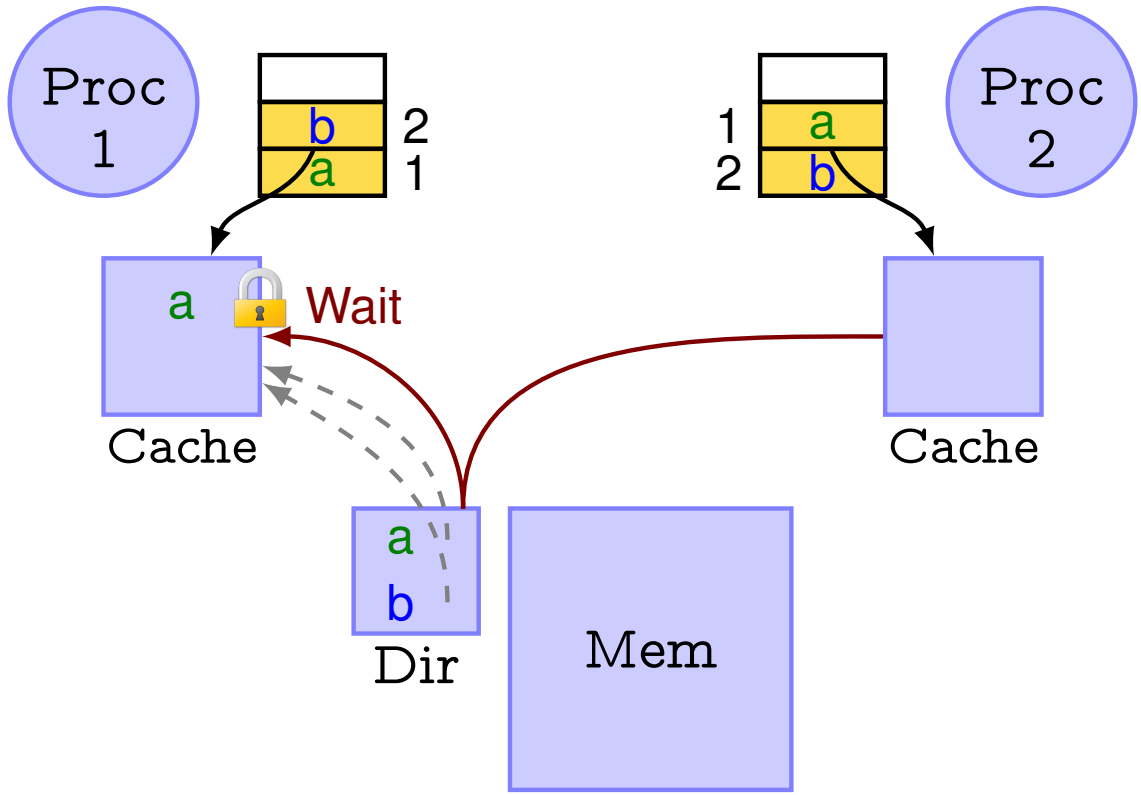


A “conflict” between atomic groups always happens in their minimun common address

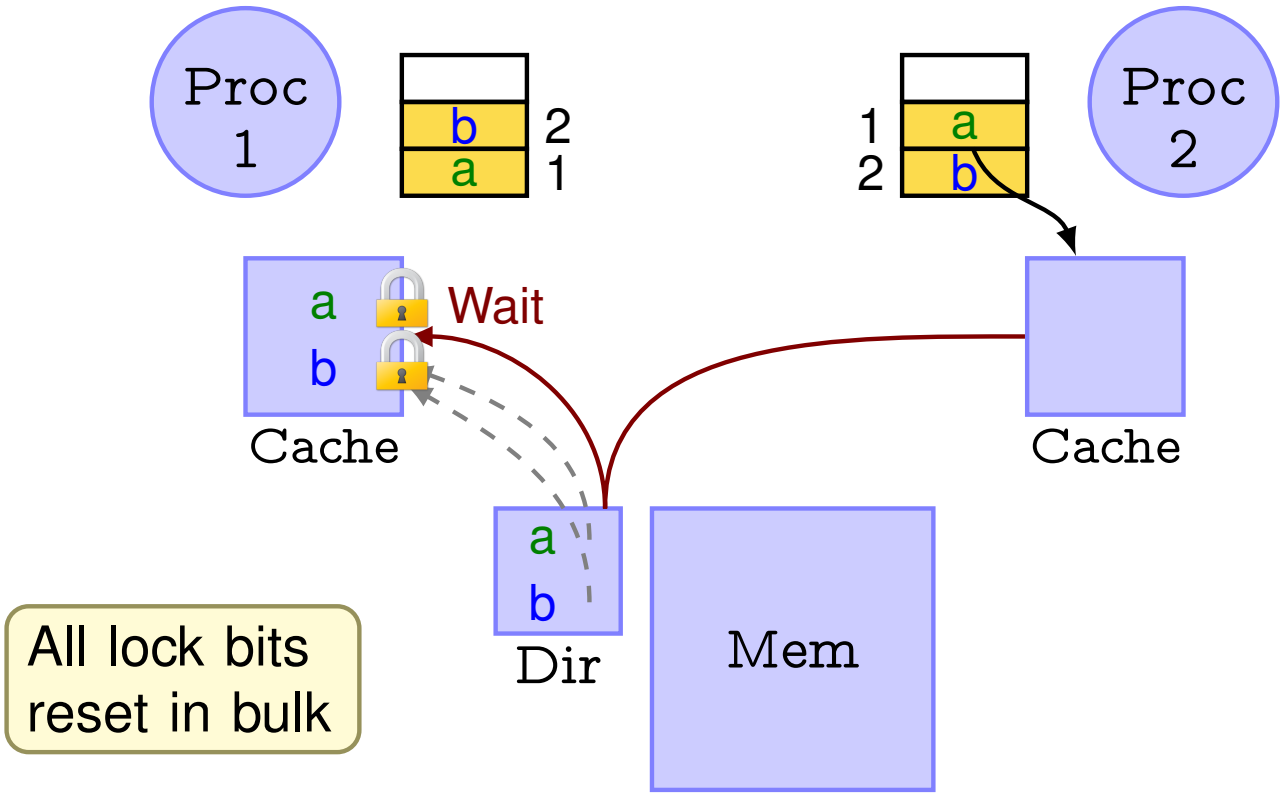
LEXICOGRAPHICAL ORDER



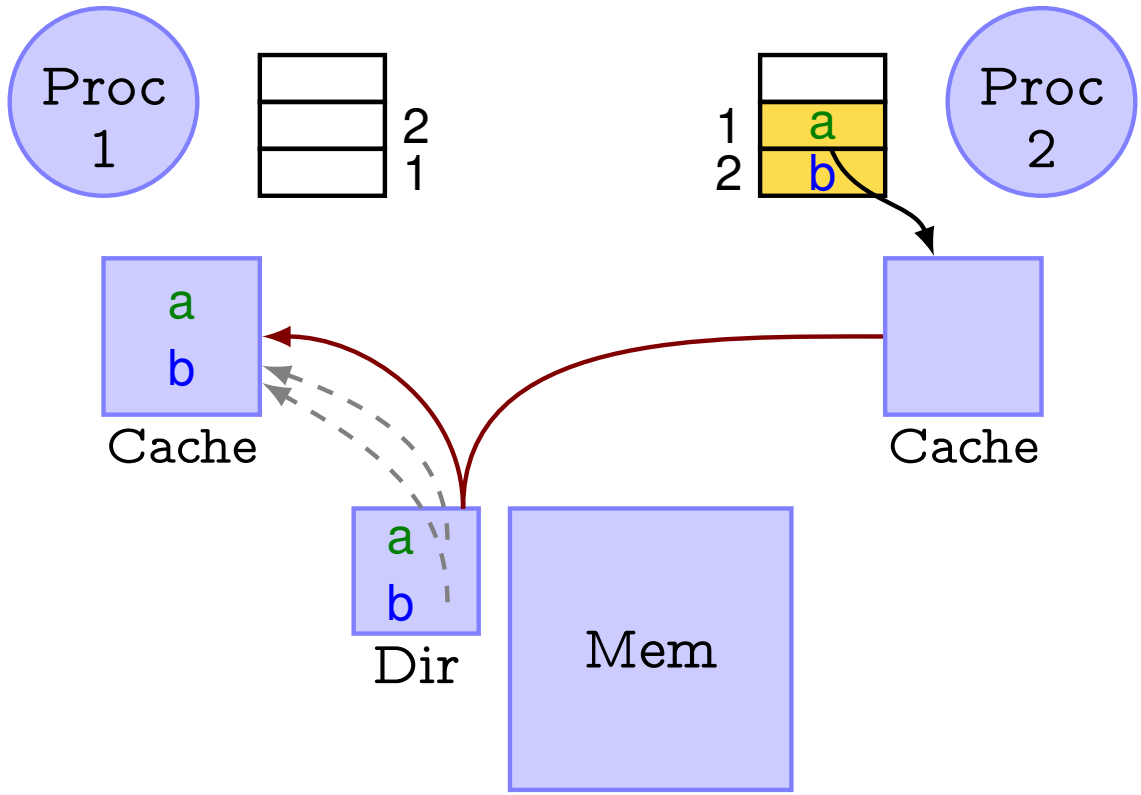
LEXICOGRAPHICAL ORDER



LEXICOGRAPHICAL ORDER



LEXICOGRAPHICAL ORDER



THE IMPACT OF THE LEX ORDER

- Used to write in cache without write permission

Cebrian, Jarhe & Ros [MICRO'24]

- Used to execute concurrently and non-speculatively multi-address atomics

Gómez-Hernández, Titos-Gil, Cebrian, Kaxiras & Ros [MICRO'21]

- Used to execute concurrently and non-speculatively critical sections

Gómez-Hernández, Cebrian, Kaxiras & Ros [ASPLOS'24]

cleAR: Non-speculative and concurrent critical sections

- Most critical sections do not change their memory footprint on retries → Inmutables

```
tmp = table[i];
table[i] = table[j];
table[j] = tmp;
```

- Hardware can learn the memory footprint of critical sections and if it is inmutable

1. Discovery
   - First (speculative) attempt to execute
   - The processor learns about the critical section structure
   - If successful, practically no overhead over baseline
2. Re-Execution
   - Only if discovery fails
   - If immutable, execute non-speculatively

AND WHAT ABOUT STRONGER CONSISTENCY

- Memory consistency models dilema: *Ros & Kaxiras* [MICRO'20]
  → Programmer intuition versus performance

AND WHAT ABOUT STRONGER CONSISTENCY

- Memory consistency models dilema: *Ros & Kaxiras* [MICRO'20]
  → Programmer intuition versus performance

- We consider two strong memory models

load→load load→store store→load store→store

AND WHAT ABOUT STRONGER CONSISTENCY

- Memory consistency models dilema: *Ros & Kaxiras* [MICRO'20]
  → Programmer intuition versus performance

- We consider two strong memory models

  load→load load→store store→load store→store

- x86-TSO: A core can see its own stores before other cores
  → Through *store-to-load forwarding*

AND WHAT ABOUT STRONGER CONSISTENCY

- Memory consistency models dilema: *Ros & Kaxiras* [MICRO'20]
  → Programmer intuition versus performance

- We consider two strong memory models

  load→load load→store store→load store→store

- x86-TSO: A core can see its own stores before other cores
  → Through *store-to-load forwarding*

- IBM 370: All cores see all stores atomically
  → More intuitive but less performant (forwarding restrictions)

AND WHAT ABOUT STRONGER CONSISTENCY

- Memory consistency models dilema: *Ros & Kaxiras* [MICRO'20]
  → Programmer intuition versus performance

- We consider two strong memory models

  load→load load→store store→load store→store

- x86-TSO: A core can see its own stores before other cores
  → Through *store-to-load forwarding*

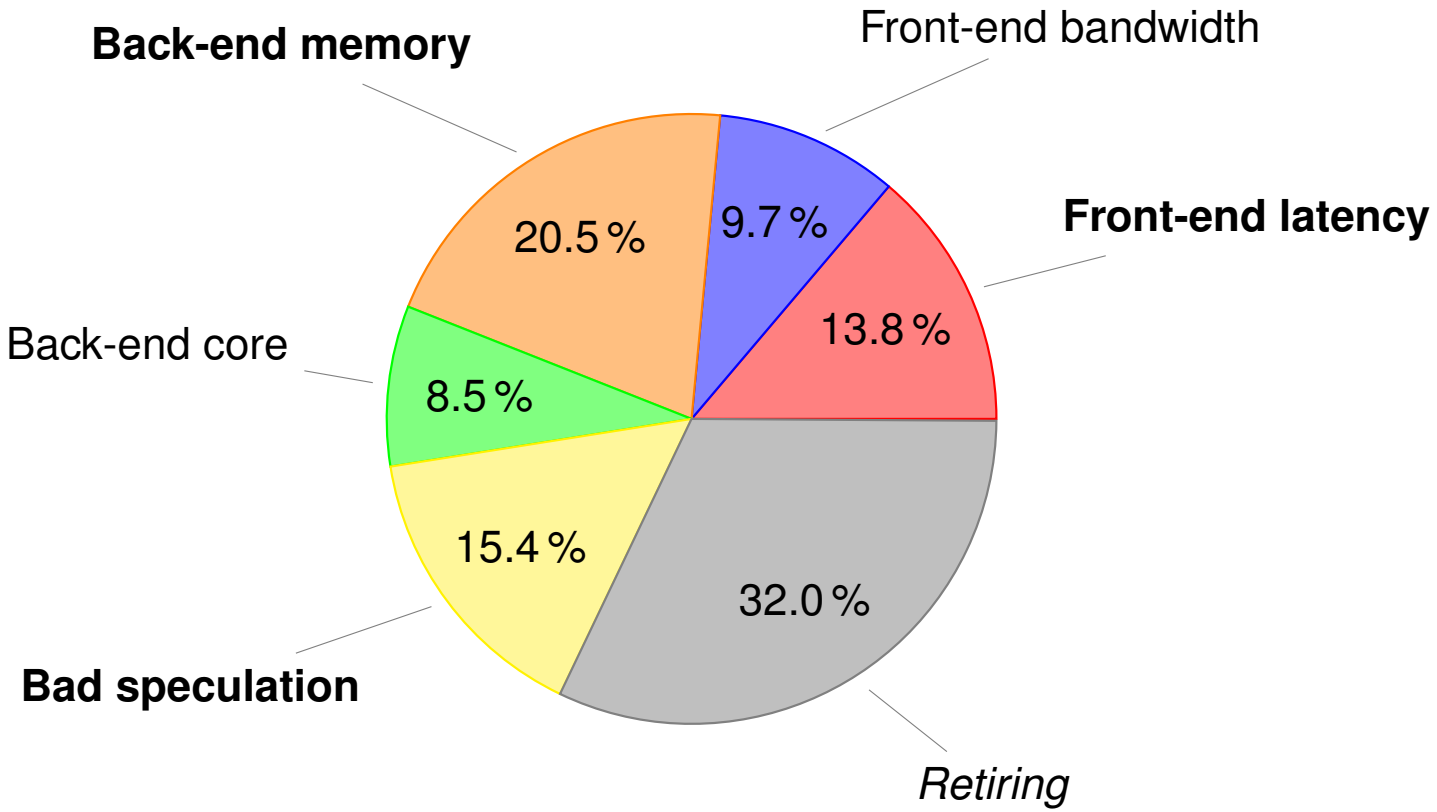
- IBM 370: All cores see all stores atomically
  → More intuitive but less performant (forwarding restrictions)

OUR SOLUTION

Both intuition and performance → load forwarded remains speculative until the store writes

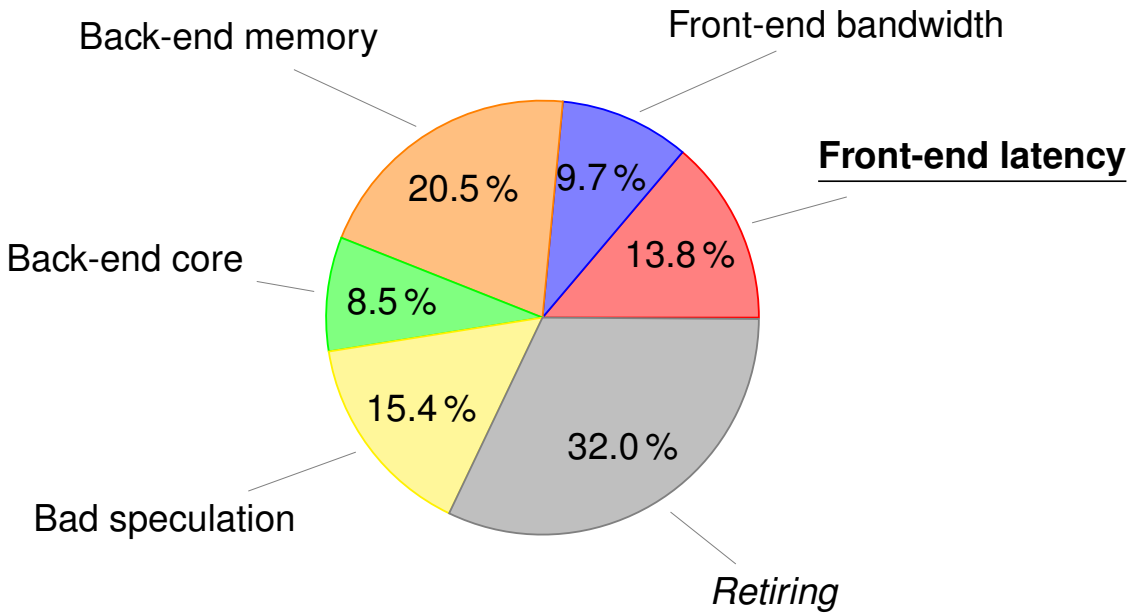
MICROARCHITECTURE

FACT: SINGLE-THREAD PERFORMANCE MATTERS



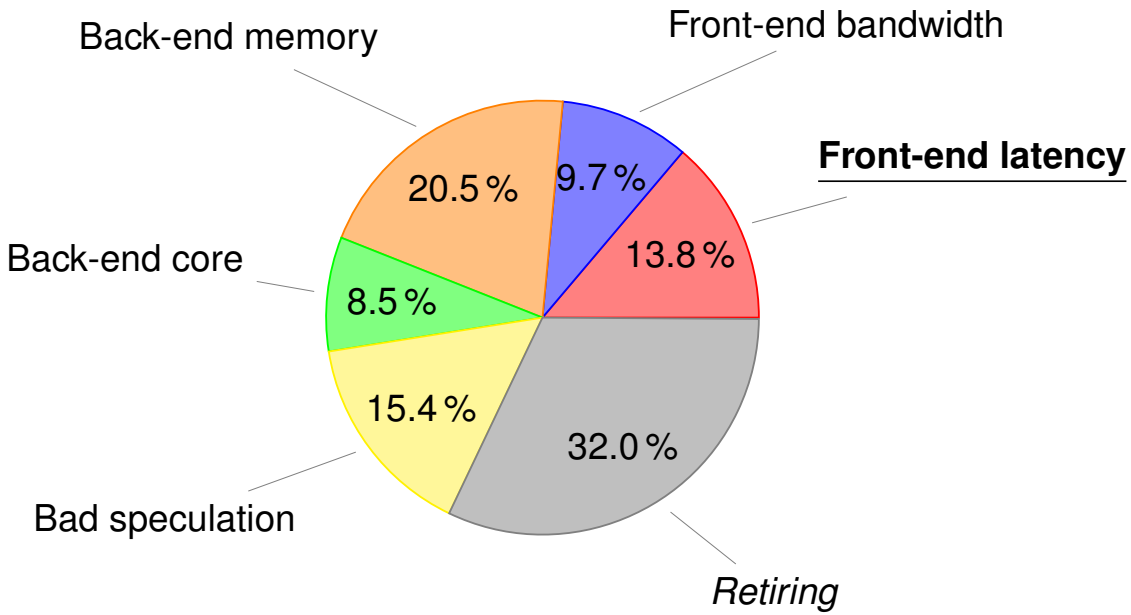
Data source: Ayers et al. *AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers*, ISCA 2019.

ACCURATE PREDICTIONS IS ALL YOU NEED



- Dominated by instruction cache (L1I) misses
  - Server and cloud apps getting larger, far from fitting in L1I
  - Hiting in the L2 or L3
- Critical! Processors need to keep the pipeline full

ACCURATE PREDICTIONS IS ALL YOU NEED



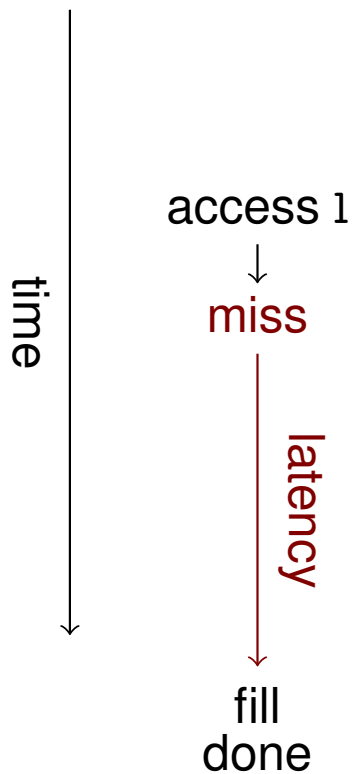
- Dominated by instruction cache (L1I) misses
  - Server and cloud apps getting larger, far from fitting in L1I
  - Hiting in the L2 or L3
- Critical! Processors need to keep the pipeline full

Solution: Instruction prefetching and criticality aware replacement

THE ENTANGLING INSTRUCTION PREFETCHER

CONCEPT OF ENTANGLED PAIRS

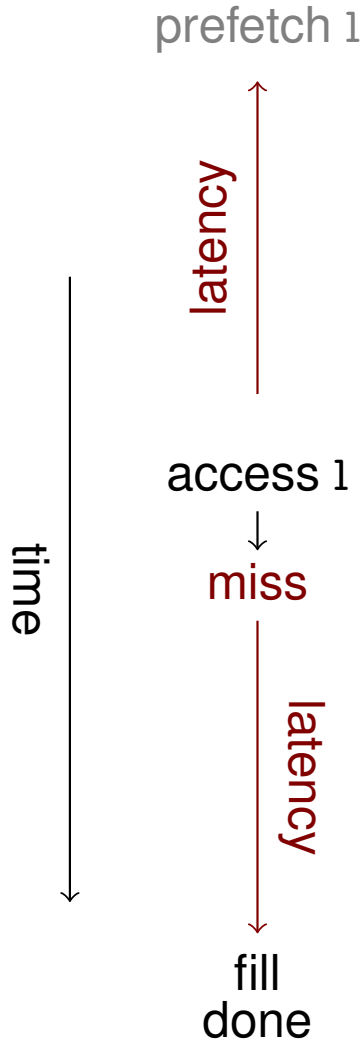
Ros & Jimborean [ISCA'21]



THE ENTANGLING INSTRUCTION PREFETCHER

CONCEPT OF ENTANGLED PAIRS

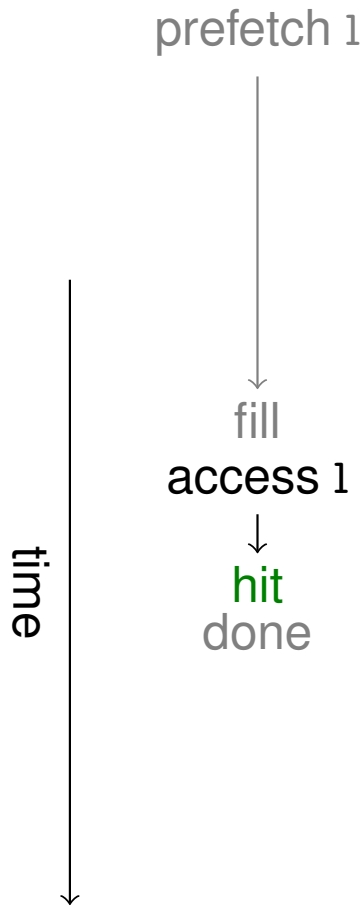
Ros & Jimborean [ISCA'21]



THE ENTANGLING INSTRUCTION PREFETCHER

CONCEPT OF ENTANGLED PAIRS

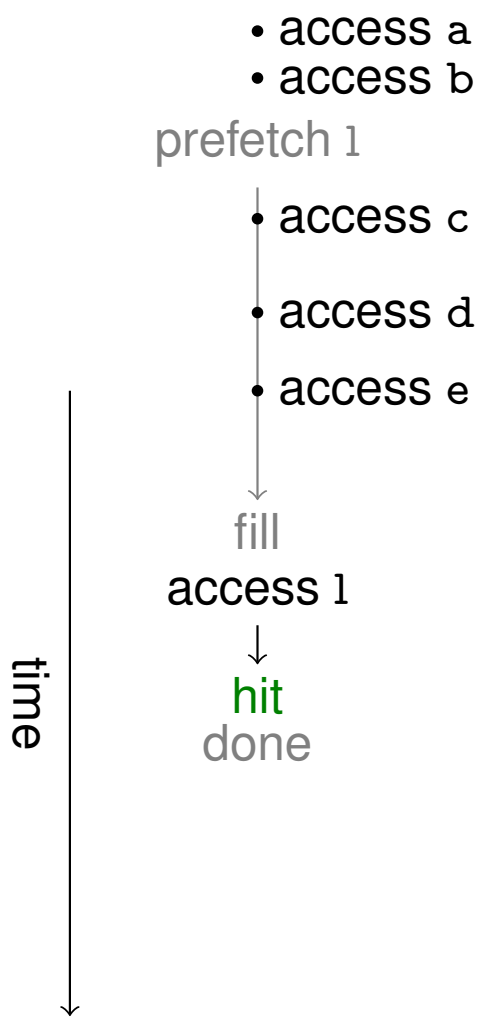
Ros & Jimborean [ISCA'21]



THE ENTANGLING INSTRUCTION PREFETCHER

CONCEPT OF ENTANGLED PAIRS

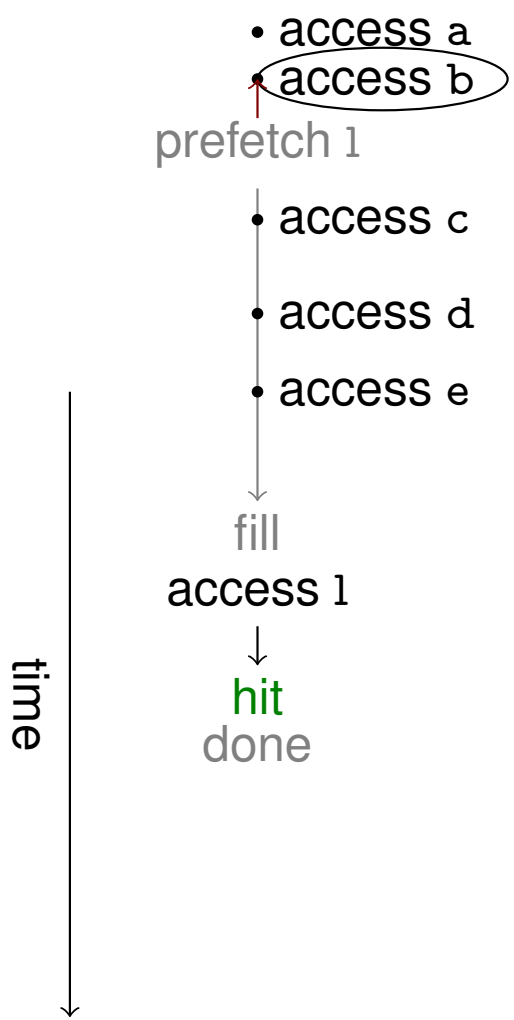
Ros & Jimborean [ISCA'21]



THE ENTANGLING INSTRUCTION PREFETCHER

CONCEPT OF ENTANGLED PAIRS

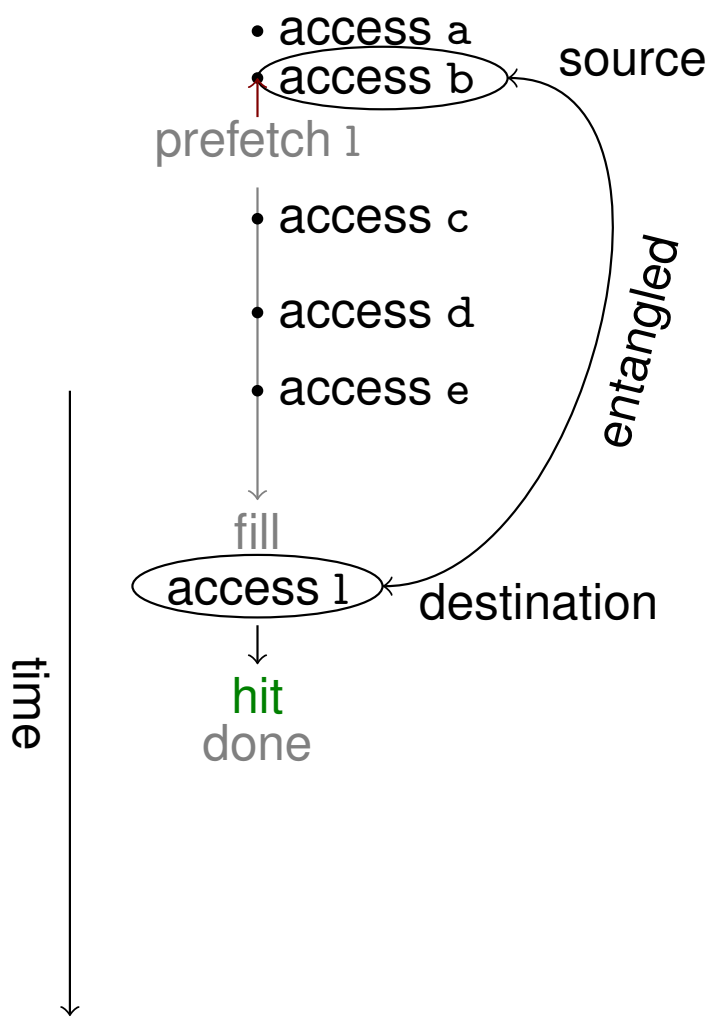
Ros & Jimborean [ISCA'21]



The Entangling Instruction Prefetcher

Concept of entangled pairs

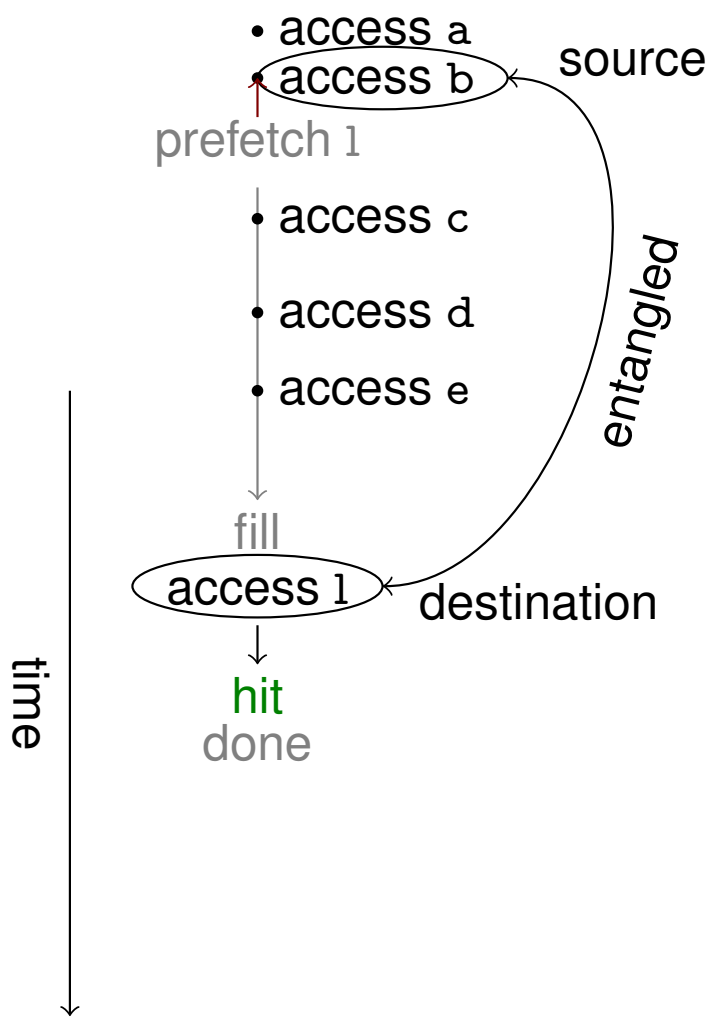
Ros & Jimborean [ISCA'21]

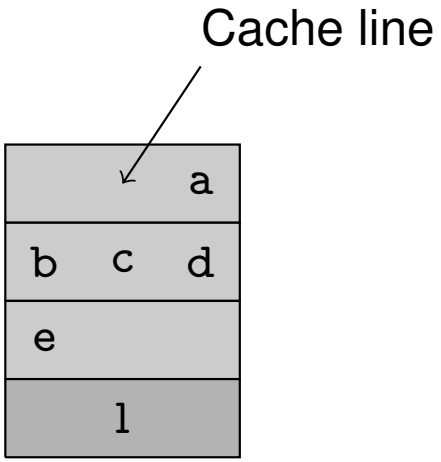


THE ENTANGLING INSTRUCTION PREFETCHER

CONCEPT OF ENTANGLED PAIRS

Ros & Jimborean [ISCA'21]

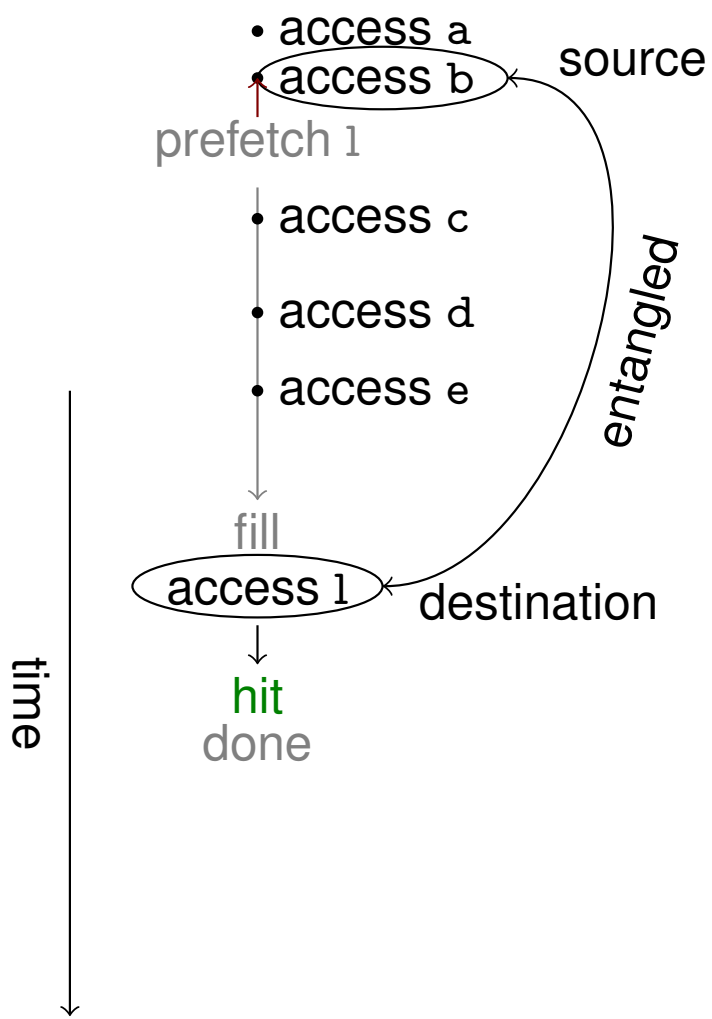


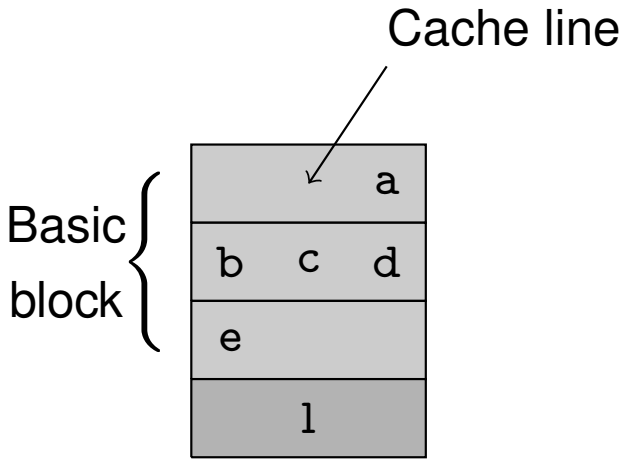


THE ENTANGLING INSTRUCTION PREFETCHER

CONCEPT OF ENTANGLED PAIRS

Ros & Jimborean [ISCA'21]

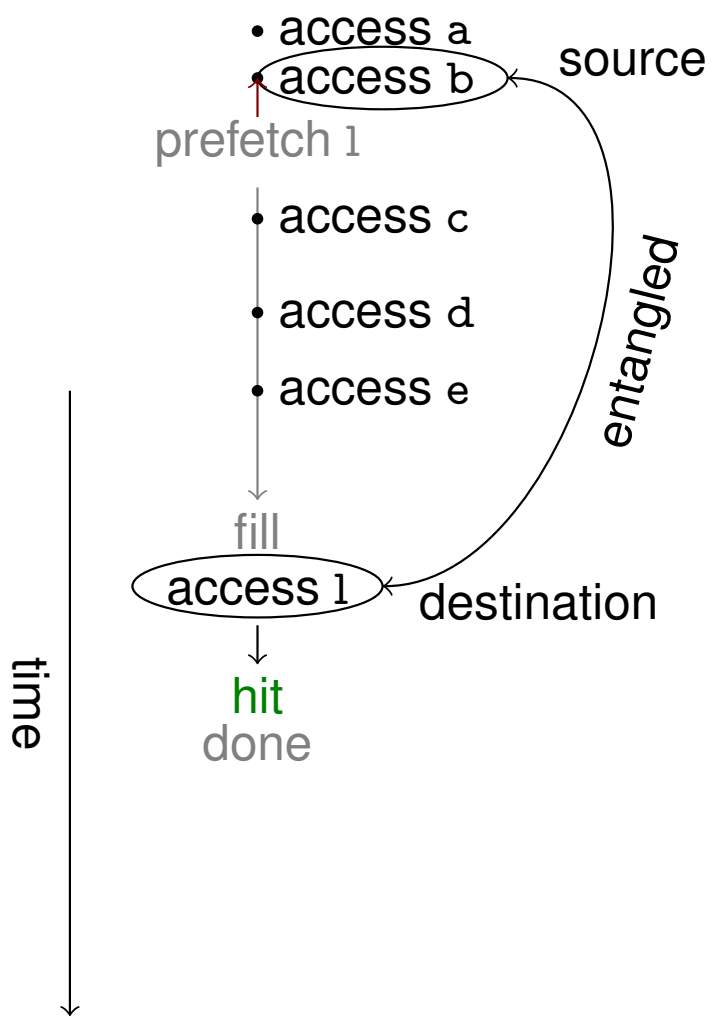


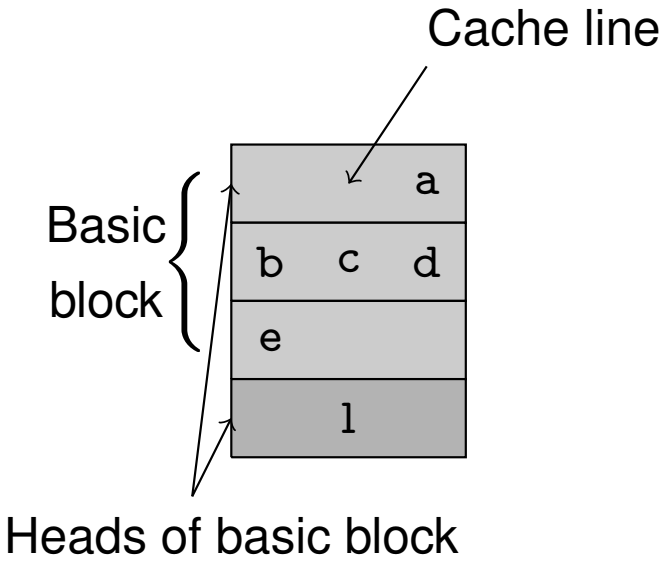


THE ENTANGLING INSTRUCTION PREFETCHER

CONCEPT OF ENTANGLED PAIRS

Ros & Jimborean [ISCA'21]

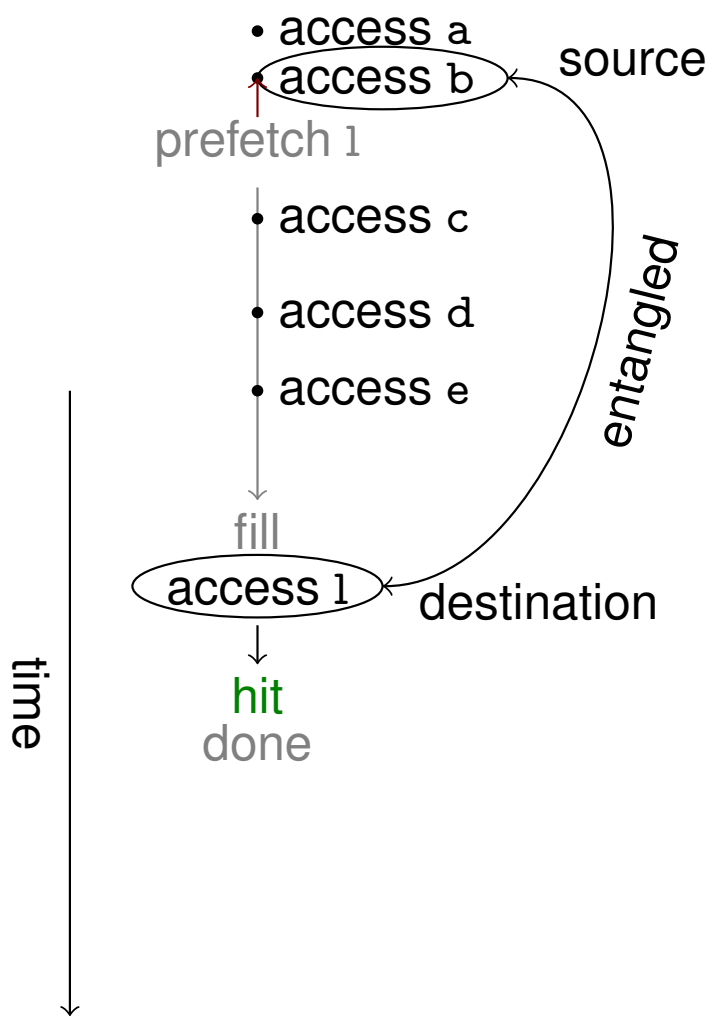


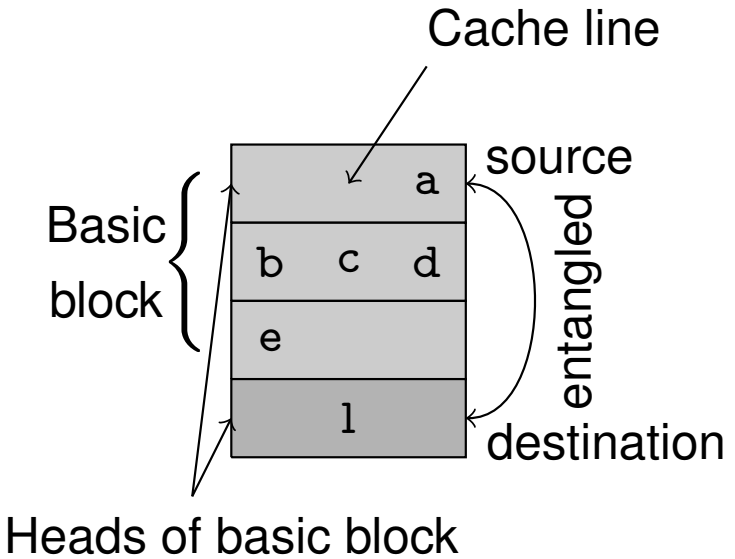


THE ENTANGLING INSTRUCTION PREFETCHER

CONCEPT OF ENTANGLED PAIRS

Ros & Jimborean [ISCA'21]

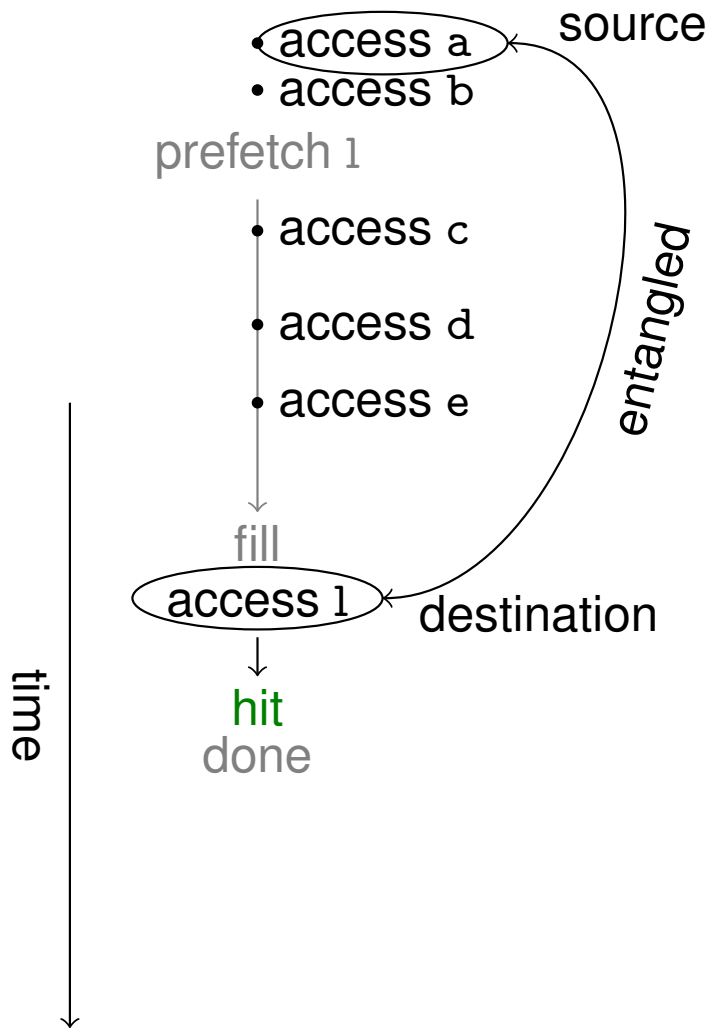


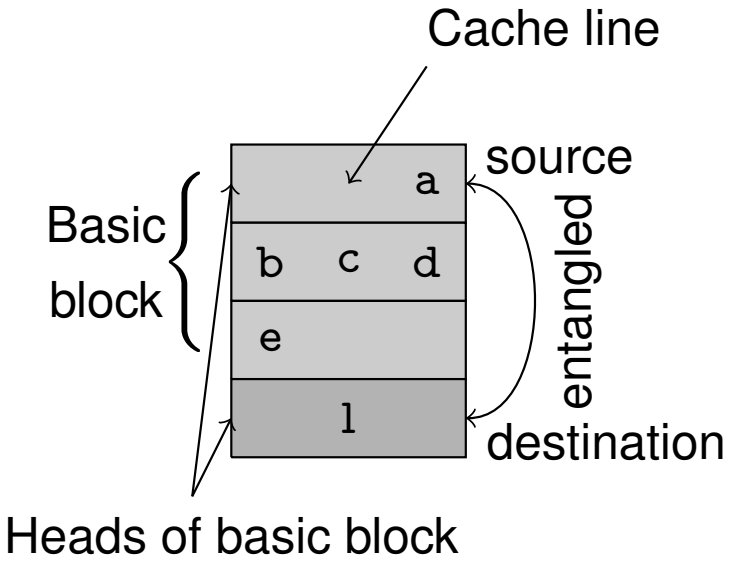


THE ENTANGLING INSTRUCTION PREFETCHER

CONCEPT OF ENTANGLED PAIRS

Ros & Jimborean [ISCA'21]





ICARUS: KEEP CRITICAL INSTRUCTIONS CLOSE

CRITICALITY AND REUSE BASED INSTRUCTION CACHING

Kalbande, Deshmukh, Ros & Panda [ASPLOS'26]

- Not all instructions are critical
  - ⇒ Most instructions can tolerate large latencies
  - ⇒ Our study with gem5 and data-center workloads show 3.5% average of critical fetches

ICARUS: KEEP CRITICAL INSTRUCTIONS CLOSE

CRITICALITY AND REUSE BASED INSTRUCTION CACHING

Kalbande, Deshmukh, Ros & Panda [ASPLOS'26]

- Not all instructions are critical
  - ⇒ Most instructions can tolerate large latencies
  - ⇒ Our study with gem5 and data-center workloads show 3.5% average of critical fetches
- OBSERVATIONS
  - Criticality is not a property of the instruction
  - Criticality is path sensitive

PC	Branch history	Critical
0xFFFFE8037D00	**0**001010	Always
	1001010	Never

(Rightmost bit denotes the recent branch prediction)

ICARUS: KEEP CRITICAL INSTRUCTIONS CLOSE

CRITICALITY AND REUSE BASED INSTRUCTION CACHING

Kalbande, Deshmukh, Ros & Panda [ASPLOS'26]

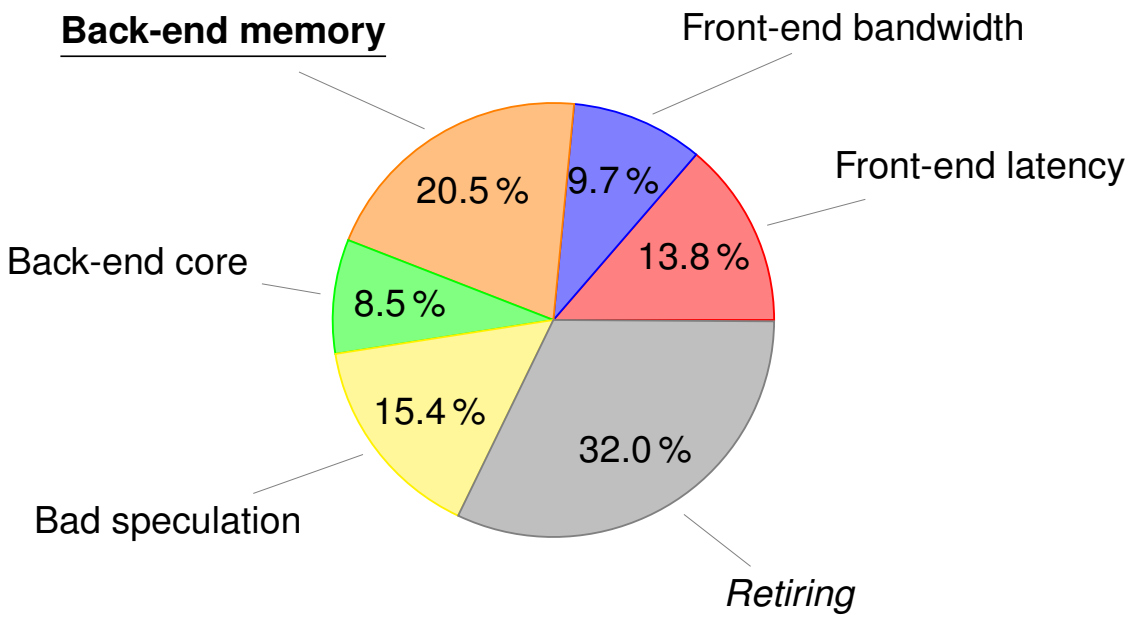
- Not all instructions are critical
  - ⇒ Most instructions can tolerate large latencies
  - ⇒ Our study with gem5 and data-center workloads show 3.5% average of critical fetches
- OBSERVATIONS
  - Criticality is not a property of the instruction
  - Criticality is path sensitive

PC	Branch history	Critical
0xFFFFE8037D00	**0**001010	Always
	1001010	Never

(Rightmost bit denotes the recent branch prediction)

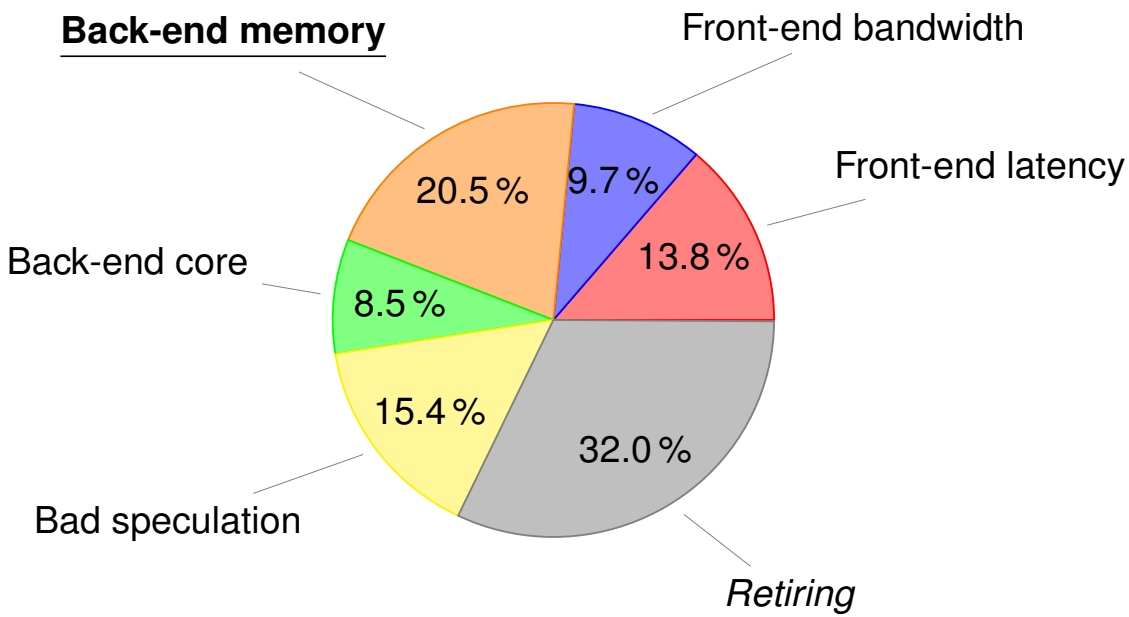
  - But cacheline reuse is also an important factor

ACCURATE PREDICTIONS IS ALL YOU NEED



- Due to data cache (L1D) misses
  - Many of them reaching main memory
- Cause stalls and late detection of bad speculation

ACCURATE PREDICTIONS IS ALL YOU NEED



- Due to data cache (L1D) misses
  - Many of them reaching main memory
- Cause stalls and late detection of bad speculation

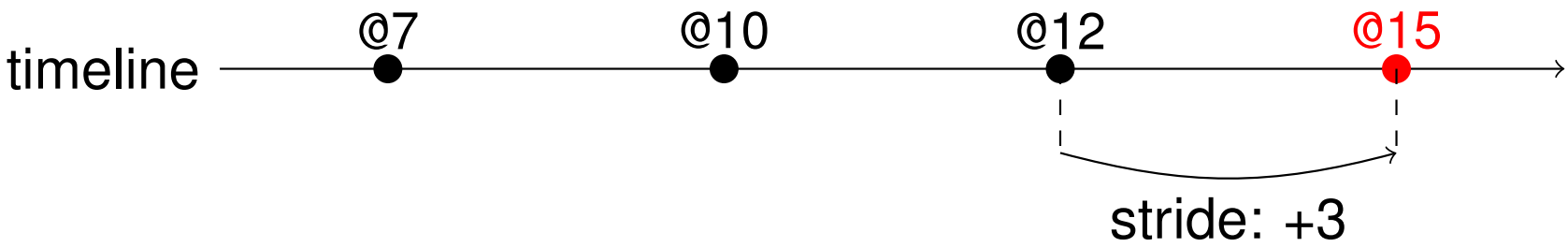
Solution: Timely data prefetching

BERTI: ACCURATE AND TIMELY LOCAL DELTA L1D PREFETCHER

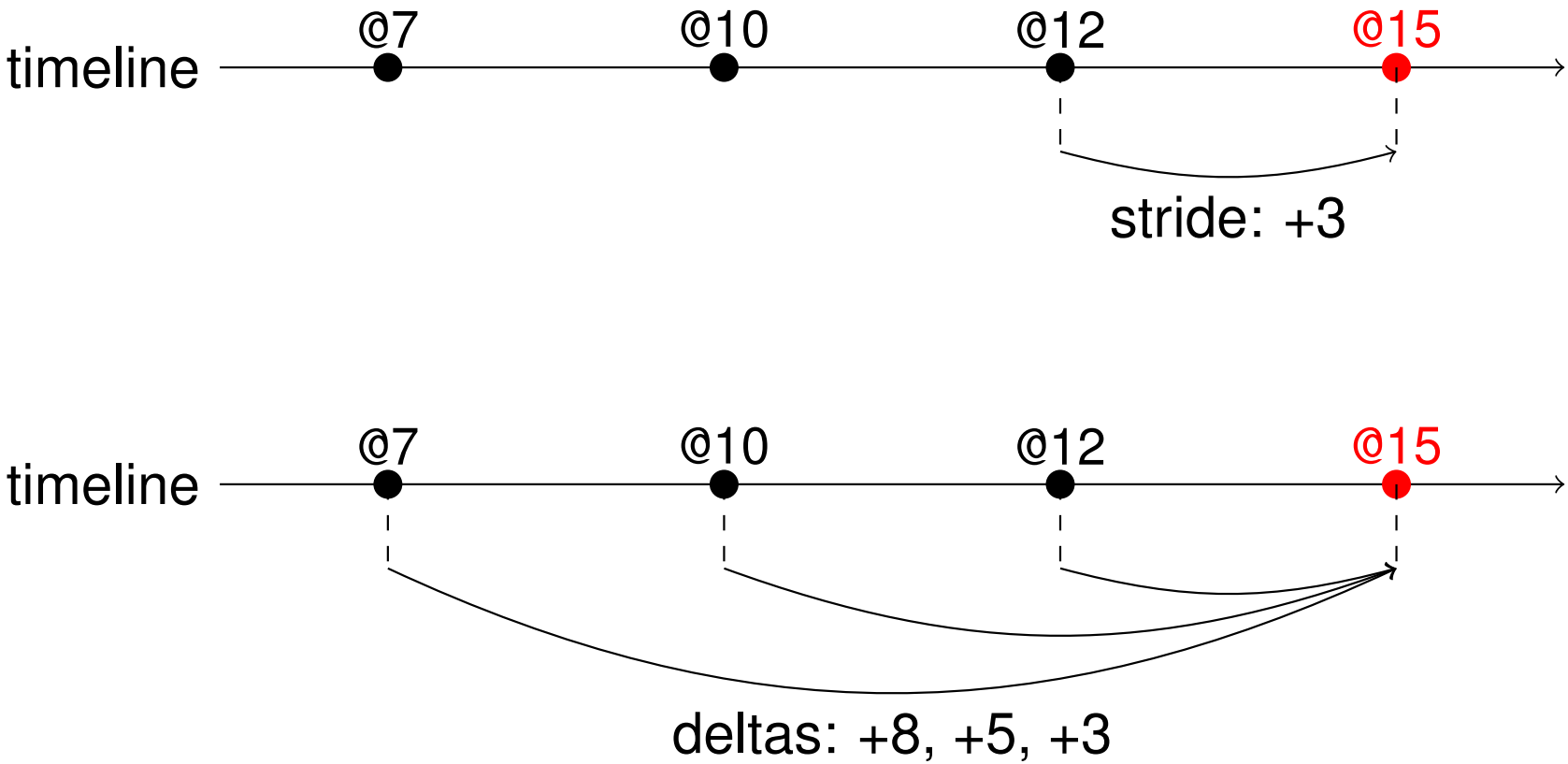
Navarro-Torres, Panda, Alastruey-Benedé, Ibañez, Viñals & Ros [MICRO'22]

- An L1D prefetcher that orchestrates data prefetching from all cache levels
- Advantages of L1D prefetching
  - Training with all processor references
  - Using the Instruction Pointer (IP) information
  - Operating with virtual addresses: cross-page prefetching

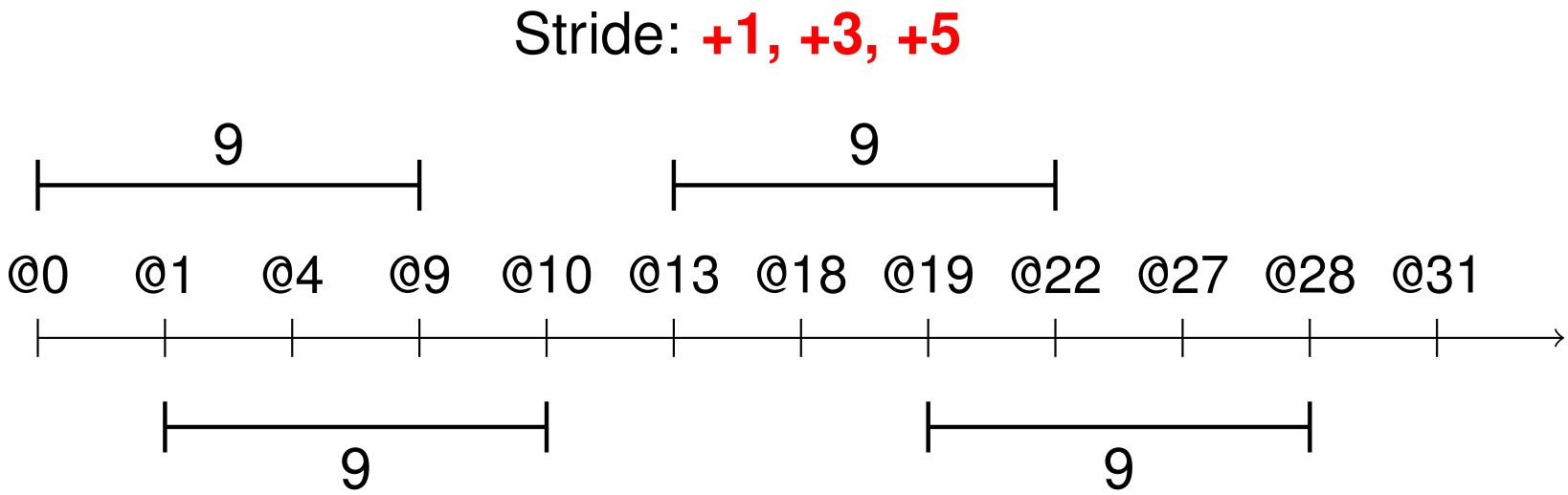
Definition of delta



Definition of delta

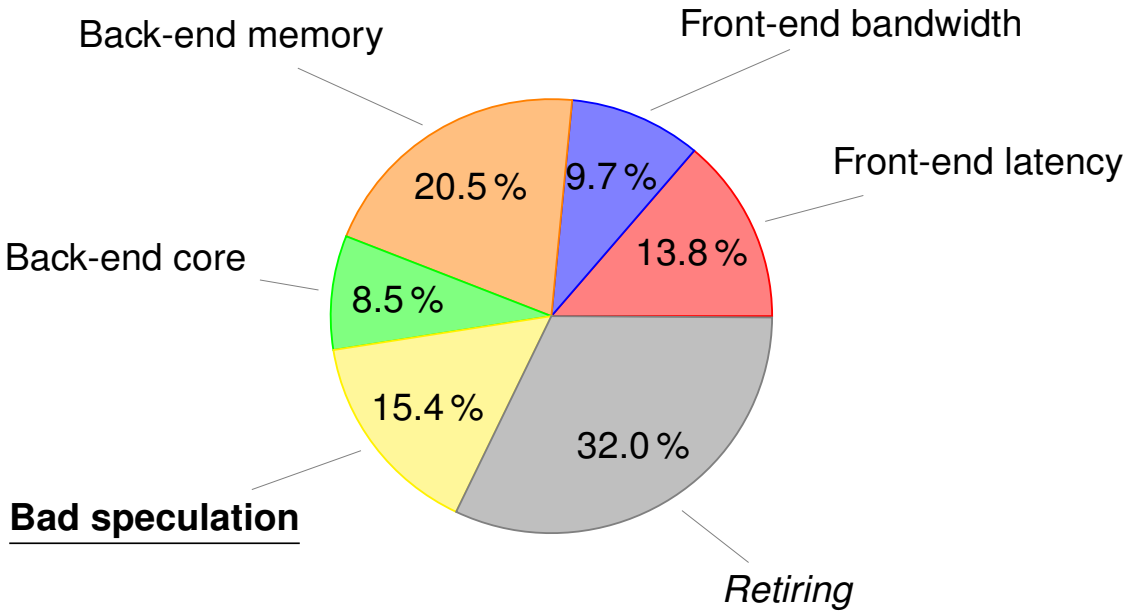


BERTI: ACCURATE AND TIMELY LOCAL DELTA L1D PREFETCHER



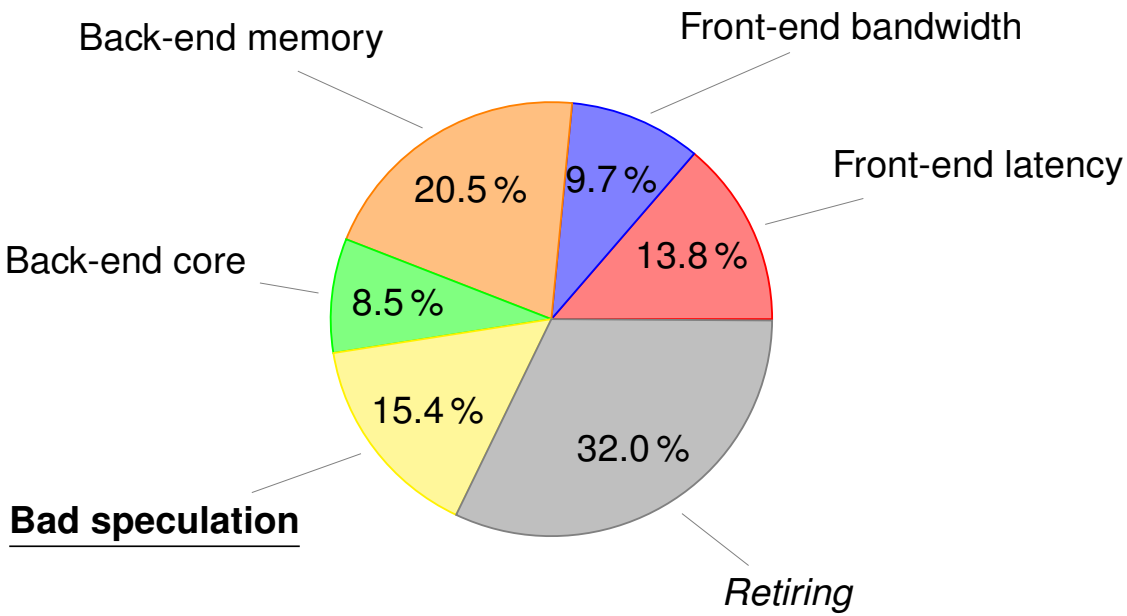
With which delta should I prefetch?
delta = 1 + 3 + 5 = 9 → **always hit**

ACCURATE PREDICTIONS IS ALL YOU NEED



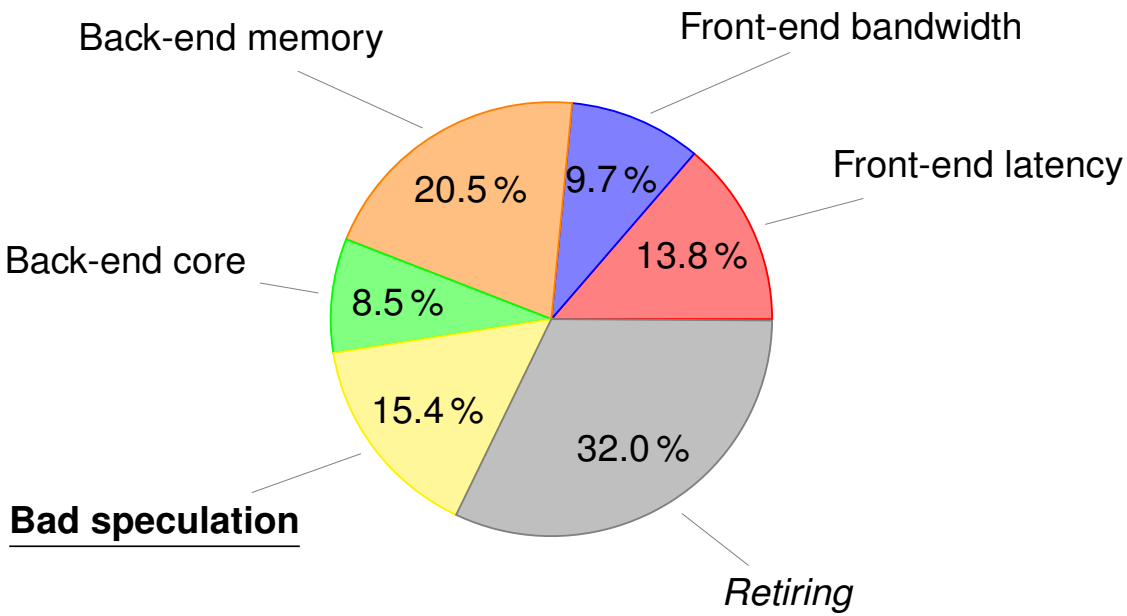
- Memory dependence prediction
- Branch prediction

ACCURATE PREDICTIONS IS ALL YOU NEED



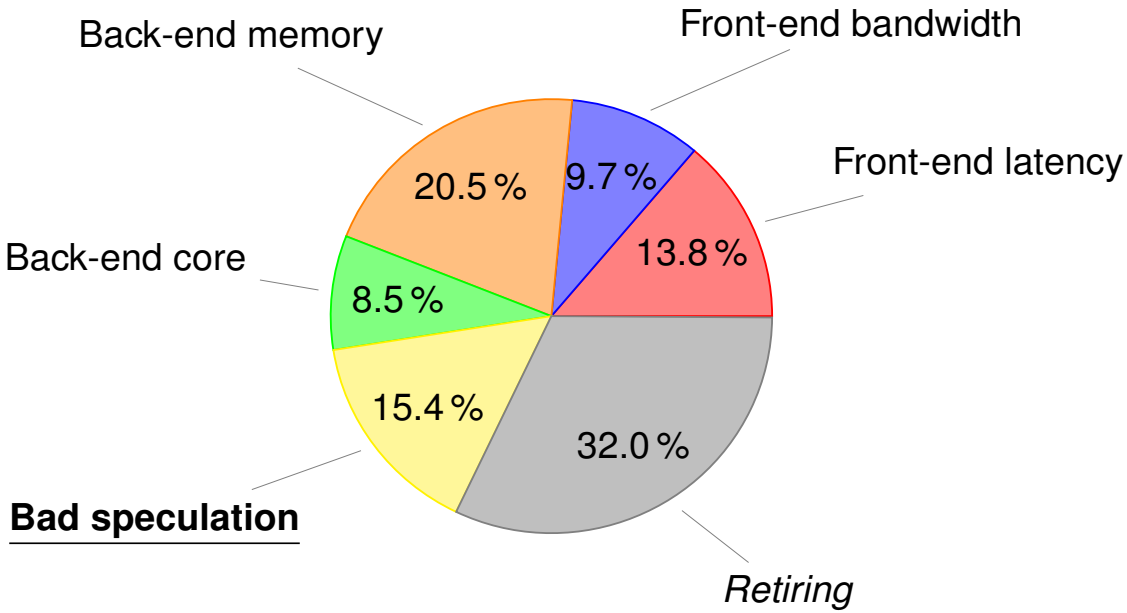
- Memory dependence prediction
  - More critical as the pipeline depth increases
- Branch prediction

ACCURATE PREDICTIONS IS ALL YOU NEED



- Memory dependence prediction
  - More critical as the pipeline depth increases
- Branch prediction
  - Hard to optimize! $\Rightarrow$ But still room for improvement

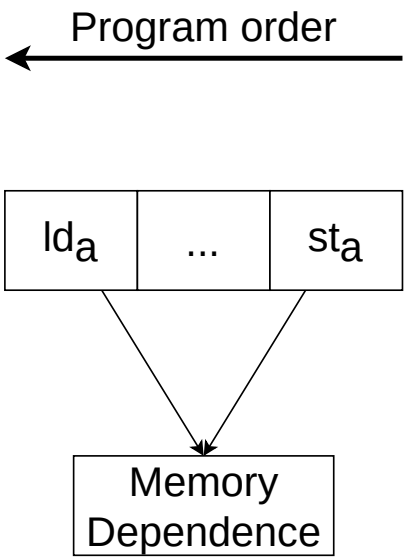
ACCURATE PREDICTIONS IS ALL YOU NEED



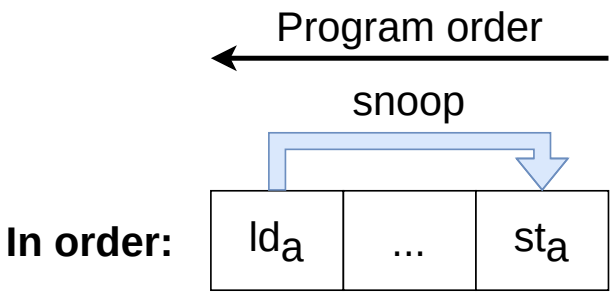
- Memory dependence prediction
  - More critical as the pipeline depth increases
- Branch prediction
  - Hard to optimize! $\Rightarrow$ But still room for improvement

Solution: Better context information and wrong-path prefetching

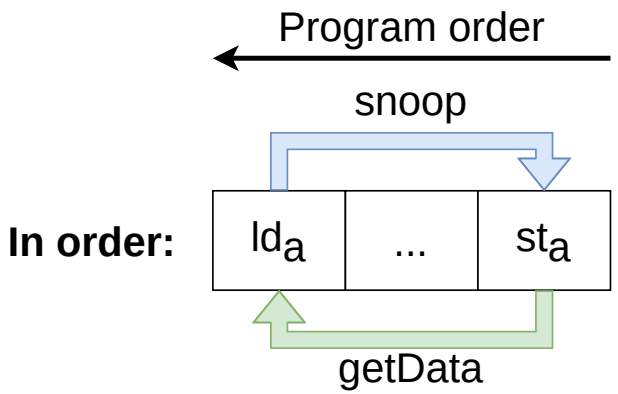
MDP: CRASH COURSE



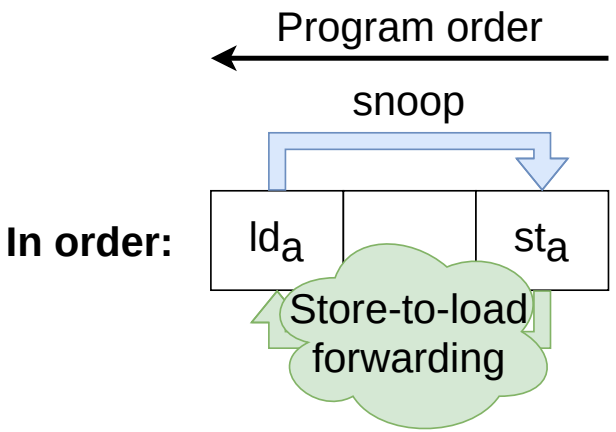
MDP: CRASH COURSE



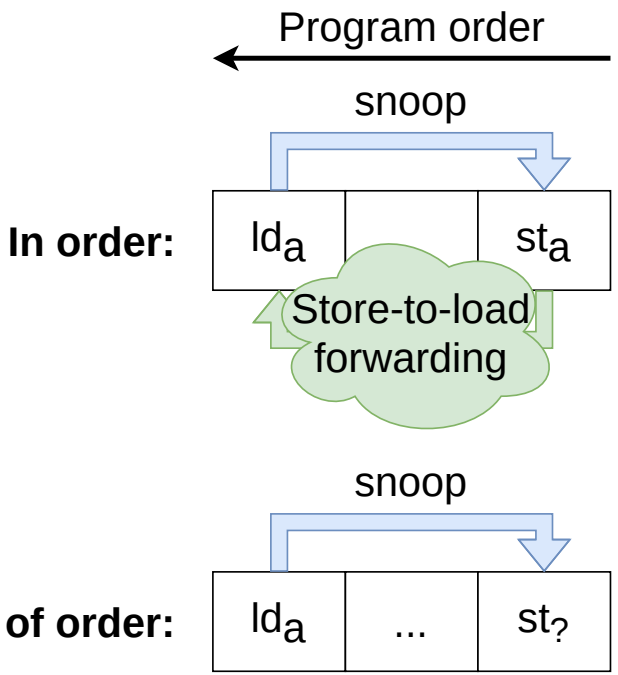
MDP: CRASH COURSE



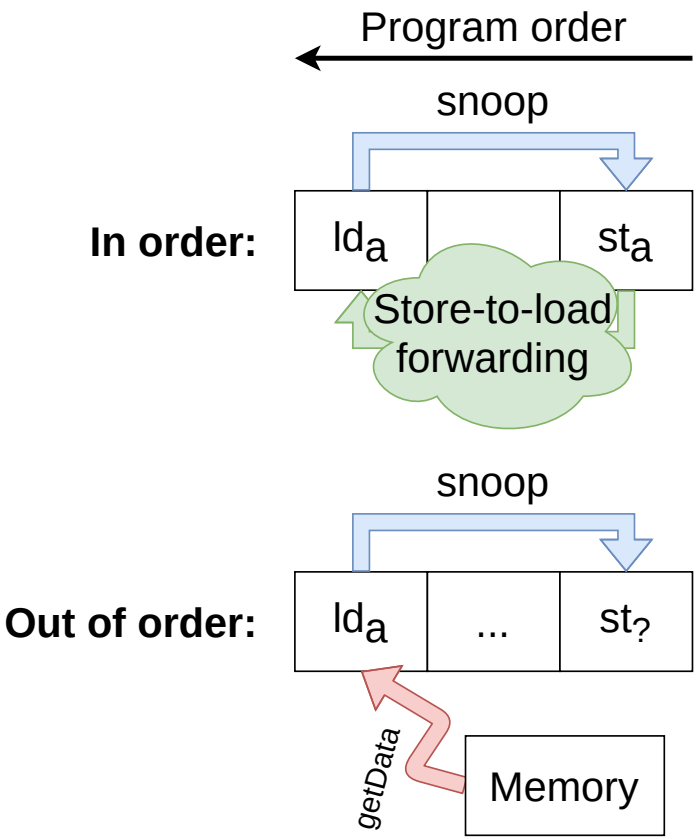
MDP: CRASH COURSE



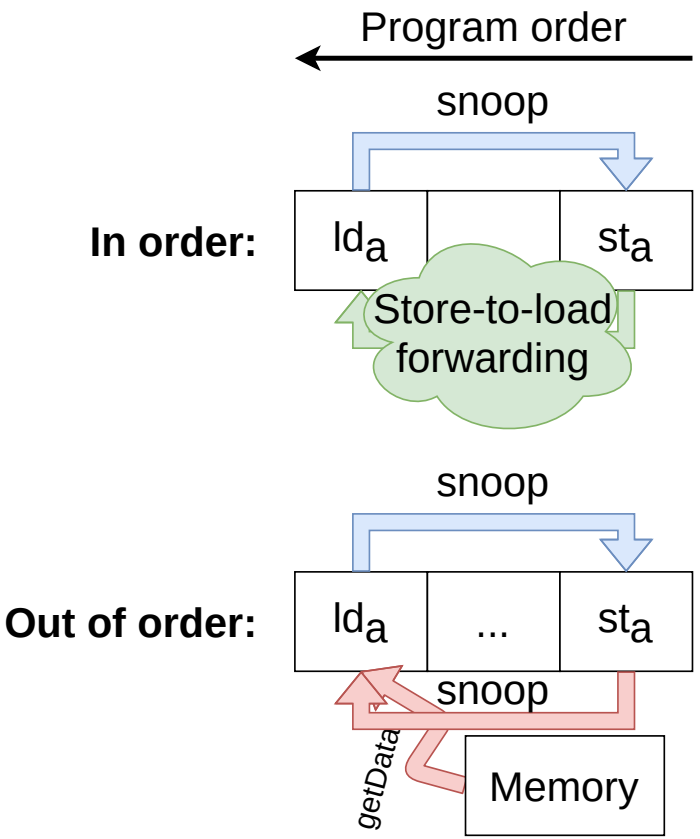
MDP: CRASH COURSE



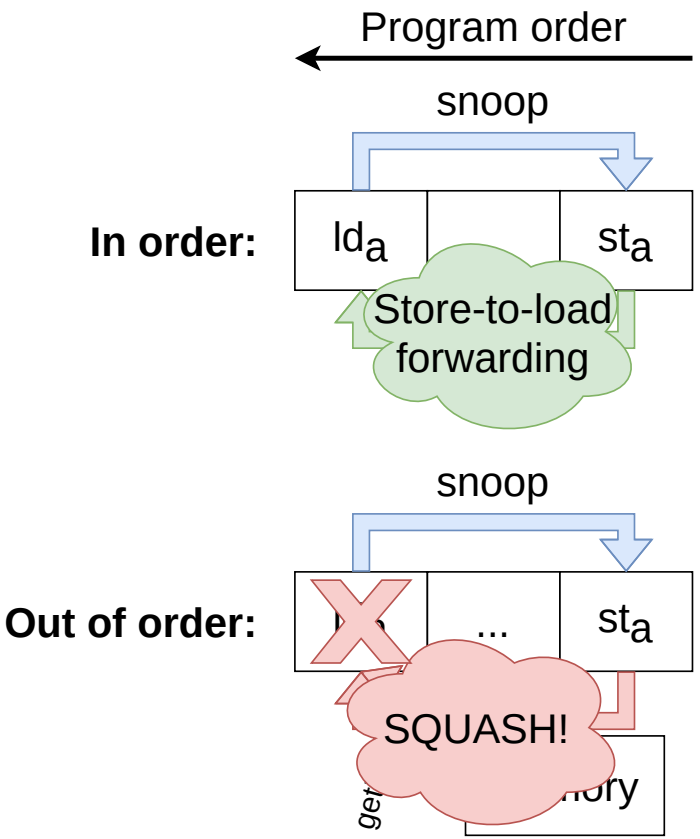
MDP: Crash course



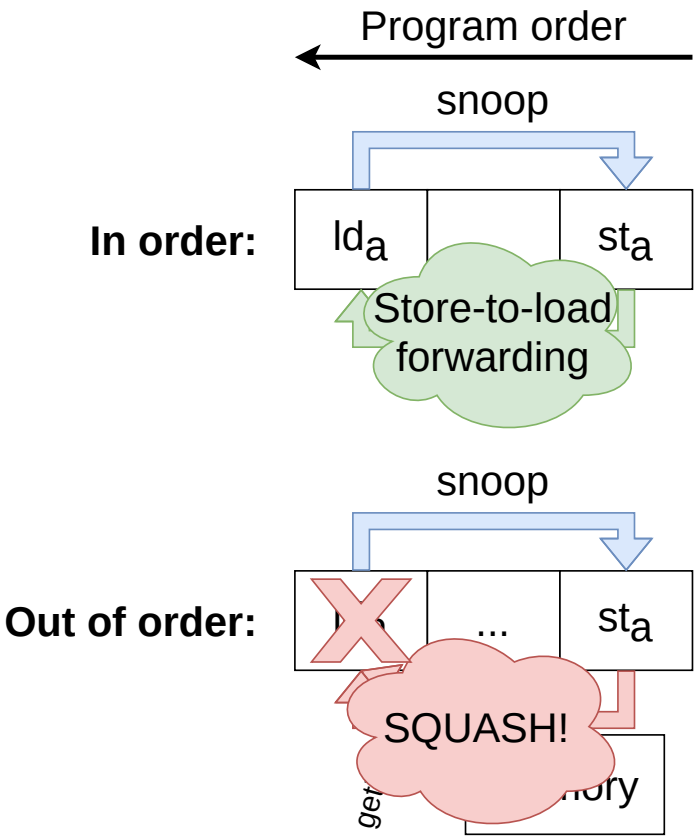
MDP: Crash course



MDP: Crash course



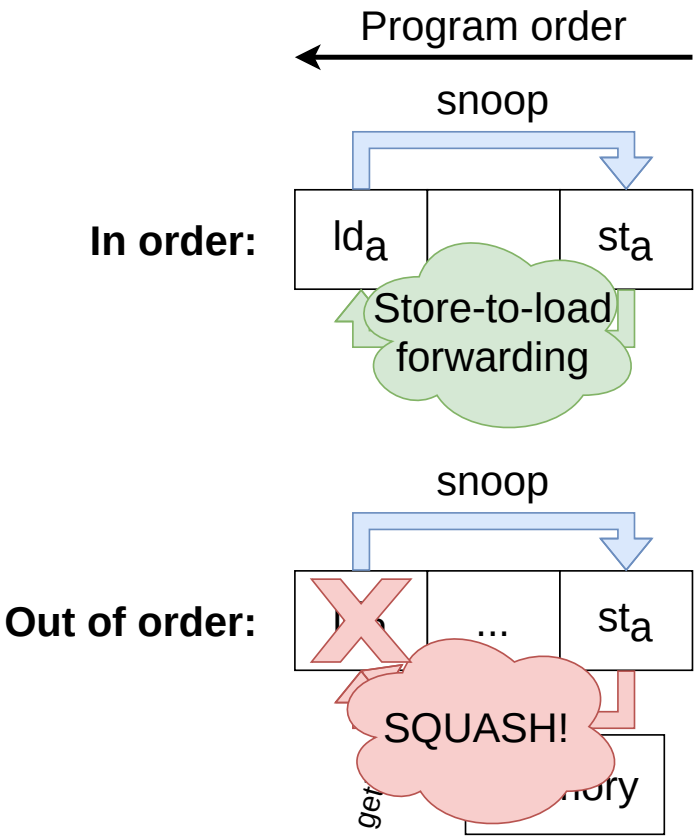
MDP: CRASH COURSE



THE PREDICTION

Is they any previous dependent store?

MDP: CRASH COURSE

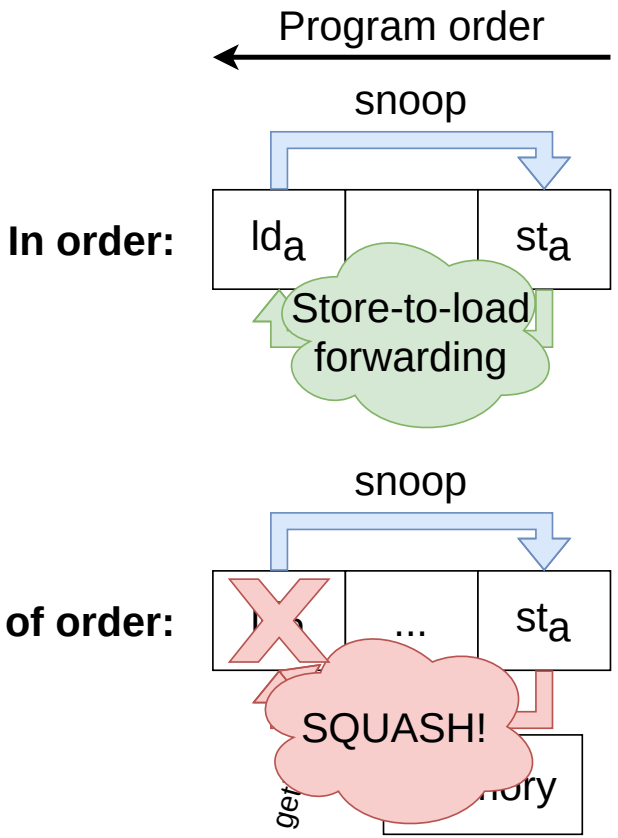


THE PREDICTION

Is they any previous dependent store?

- NO: Execute the load
- YES: Stall (wait) until it executes

MDP: CRASH COURSE



THE PREDICTION

Is they any previous dependent store?

- NO: Execute the load
- YES: Stall (wait) until it executes

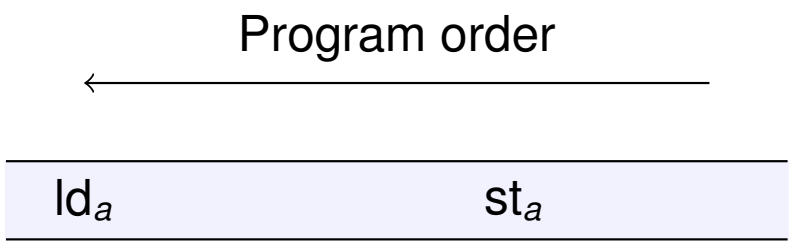
THE MISPREDICTION

- Execute, but conflict ☹
- Stall, but no dependence 😐

PHAST: MEMORY DEPENDENCE PREDICTION

KNOWING THE EXACT NEEDED CONTEXT

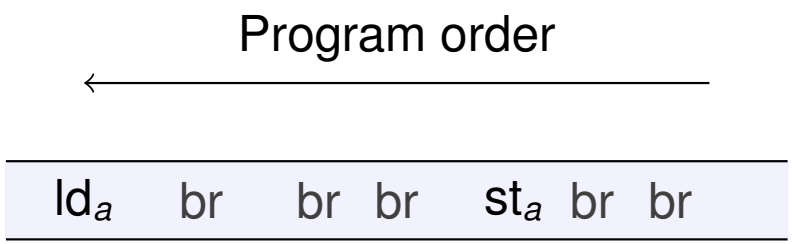
Kim & Ros [HPCA'24]



PHAST: MEMORY DEPENDENCE PREDICTION

KNOWING THE EXACT NEEDED CONTEXT

Kim & Ros [HPCA'24]



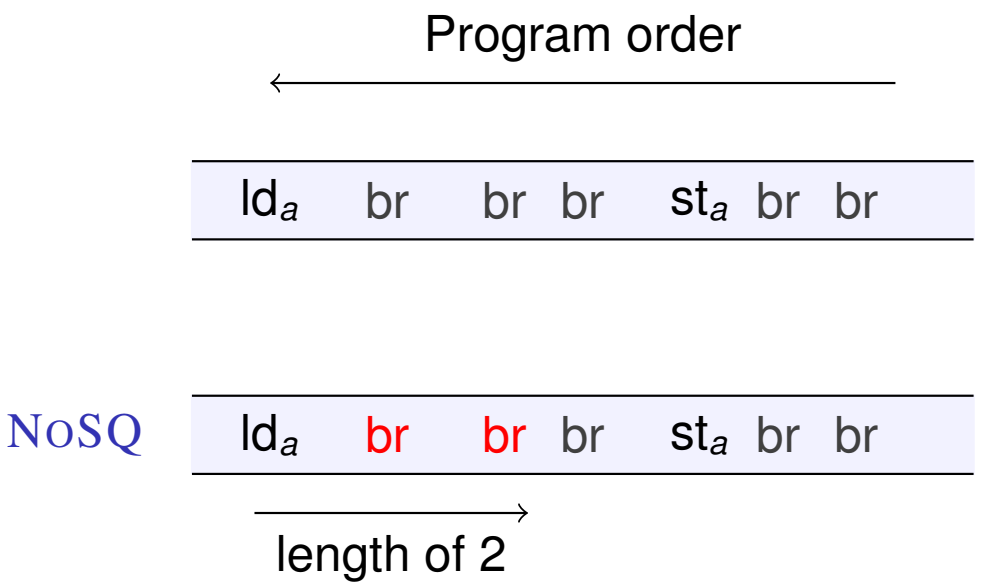
Use load's branch history

But which history length?

PHAST: MEMORY DEPENDENCE PREDICTION

KNOWING THE EXACT NEEDED CONTEXT

Kim & Ros [HPCA'24]



Use load's branch history

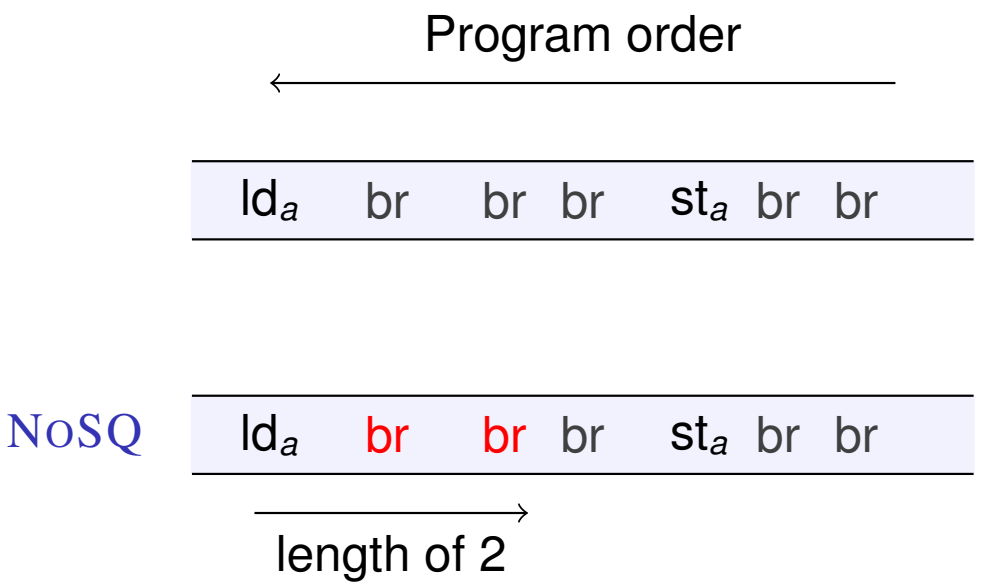
But which history length?

Predetermined history length

PHAST: MEMORY DEPENDENCE PREDICTION

KNOWING THE EXACT NEEDED CONTEXT

Kim & Ros [HPCA'24]



Use load's branch history

But which history length?

Predetermined history length

Too short? Mispredictions

Too long? Unnecessary paths

PHAST: MEMORY DEPENDENCE PREDICTION

KNOWING THE EXACT NEEDED CONTEXT

Kim & Ros [HPCA'24]



Use load's branch history

But which history length?

Predetermined history length

Too short? Mispredictions

Too long? Unnecessary paths

MDP-TAGE

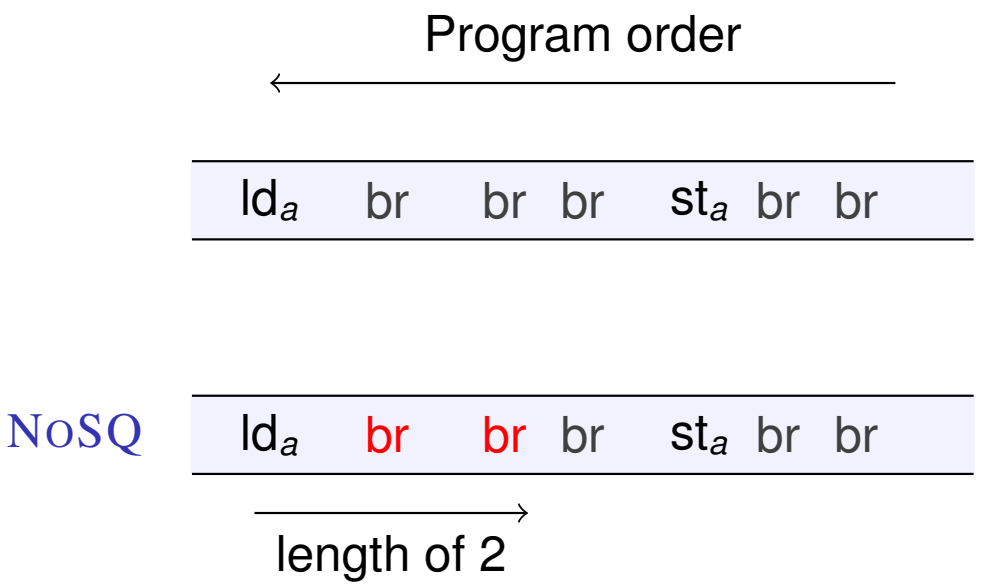
Geometrically increasing history lengths

Brute-force method to find the right lenght

PHAST: MEMORY DEPENDENCE PREDICTION

KNOWING THE EXACT NEEDED CONTEXT

Kim & Ros [HPCA'24]



Use load's branch history

But which history length?

Predetermined history length

Too short? Mispredictions

Too long? Unnecessary paths

MDP-TAGE

Geometrically increasing history lengths

Brute-force method to find the right lenght

- Previous context-sensitive proposals borrow branch-prediction techniques
  - Use predetermined history lengths ⇒ sub-optimal

PHAST: Memory dependence prediction

Knowing the exact needed context

- PHAST uses the proper history length for each conflict
  - $\Rightarrow$ The one that defines the path from the store to the load
    - Intuition: If the exact execution path repeats, it is likely that the dependence repeats, too

PHAST: Memory dependence prediction

Knowing the exact needed context

- PHAST uses the proper history length for each conflict
  - $\Rightarrow$ The one that defines the path from the store to the load
    - Intuition: If the exact execution path repeats, it is likely that the dependence repeats, too

- To identify the path, PHAST only considers divergent branches (conditional and indirect branches):
  - Conditional branches $\rightarrow$ taken/not taken bit
  - Indirect branches $\rightarrow$ target address

FINAL REMARKS

- A *long* journey, but many challenges remain to solve
- Opening the coherence black box to consistency can provide large benefits
- Lex order is an effective solution to *concurrency* problems
- Traditional microarchitecture still matters
- But what about AI?

By 2030, most servers will still need CPU cores, because AI systems depend on them for orchestration, data movement, preprocessing, networking, and feeding accelerators efficiently; the design goal is to make those cores far more area- and energy-efficient.

THANKS: THE CAPS RESEARCH GROUP



THANKS: THE ECHO TEAM









A Journey from Cache Coherence to Microarchitecture

Alberto Ros

University of Murcia, Spain

Thank you!



ECHO, ERC Consolidator Grant (No 819134)