Protocolos de coherencia sensibles al modelo de consistencia

Alberto Ros Manuel E. Acacio

Universidad de Murcia
{aros,meacacio}@ditec.um.es

13 de febrero de 2015

Outline

Outline

Algoritmo de Dekker

Algoritmo de Dekker

```
/* Inicialmente X = Y = 0 */

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

- ¿Puede dar el resultado X=0, Y=0?

Algoritmo de Dekker

Algoritmo de Dekker

```
/* Inicialmente X = Y = 0 */

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

- ¿Puede dar el resultado X=0, Y=0?
- Sí, si la lectura se ejecuta antes de la escritura del mismo hilo

Consistencia

- Es necesario definir un modelo de consistencia de memoria, o simplemente modelo de consistencia
- El modelo de consistencia es el contrato entre el programador y la máquina
  - El programador sabe que resultados puede esperar
  - La máquina sabe cuánto puede optimizar el programa

Consistencia

- Es necesario definir un modelo de consistencia de memoria, o simplemente modelo de consistencia
- El modelo de consistencia es el contrato entre el programador y la máquina
  - El programador sabe que resultados puede esperar
  - La máquina sabe cuánto puede optimizar el programa

Definición de modelo de consistencia

El modelo de consistencia es una especificación del comportamiento permitido en programas multihilo que se ejecutan bajo memoria compartida

Consistencia

- Es necesario definir un modelo de consistencia de memoria, o simplemente modelo de consistencia
- El modelo de consistencia es el contrato entre el programador y la máquina
  - El programador sabe que resultados puede esperar
  - La máquina sabe cuánto puede optimizar el programa

Definición de modelo de consistencia

El modelo de consistencia es una especificación del comportamiento permitido en programas multihilo que se ejecutan bajo memoria compartida

- En concreto, el modelo de consistencia especifica el valor que devolverá cada lectura y el estado final de la memoria después de la ejecución del programa

Consistencia vs. Coherencia

- NO confundir coherencia con consistencia
- La coherencia define el comportamiento de cada bloque de memoria de forma aislada mediante dos invariantes
  1. Single-Writer-Multiple-Reader (SWMR)
  2. Data-Value
- La coherencia NO define el comportamiento de la memoria compartida, simplemente hace el uso de cachés transparente al programador
- La consistencia define el comportamiento de todos los accesos a memoria a distintas posiciones
- La consistencia puede hacer uso de la coherencia

Modelos de consistencia

- Hay muchos modelos de consistencia
  - Sequential Consistency (SC)
  - Total Store Order (TSO)
  - Processor Consistency (PC)
  - Modelo de consistencia de ARM
  - Modelo de consistencia de IBM Power
  - Weak Consistency (WC)
  - Release Consistency (RC)
  - Scope Consistency (ScC)
  - Entry Consistency
- Vamos a ver sólo los más populares

Sequential consistency (SC)

Formalizada por Lamport[1]

Un multiprocesador es secuencialmente consistente si el resultado de cualquier ejecución es el mismo que si las operaciones de todos los procesadores fueran ejecutados en algún orden secuencial y las operaciones de cada procesador aparecen en esa secuencial en el orden especificado por el programa.

[1] L. Lamport, “How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs”, IEEE Transactions on Computers, Sept. 1979.

Sequential consistency (SC)

Formalizada por Lamport[1]

Un multiprocesador es secuencialmente consistente si el resultado de cualquier ejecución es el mismo que si las operaciones de todos los procesadores fueran ejecutados en algún orden secuencial y las operaciones de cada procesador aparecen en esa secuencial en el orden especificado por el programa.

- La ordenación de todos los accesos a memoria de todos los hilos se conoce como ordenación de memoria (<m)

¿Posibles ordenaciones bajo SC?

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

[1] L. Lamport, “How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs”, IEEE Transactions on Computers, Sept. 1979.

SEQUENTIAL CONSISTENCY (SC)

- Formalmente, siendo <p el orden en el programa y <m el orden en memoria:
  - Si L(a) <p L(b) $\Rightarrow$ L(a) <m L(b) /* Load$\rightarrow$Load */
  - Si L(a) <p S(b) $\Rightarrow$ L(a) <m S(b) /* Load$\rightarrow$Store */
  - Si S(a) <p S(b) $\Rightarrow$ S(a) <m S(b) /* Store$\rightarrow$Store */
  - Si S(a) <p L(b) $\Rightarrow$ S(a) <m L(b) /* Store$\rightarrow$Load */
  - Igual para las operaciones atómicas (RMW)

- Modelo empleado en el MIPS R10000

Sequential consistency (SC)

- Formalmente, siendo <p el orden en el programa y <m el orden en memoria:
  - Si L(a) <p L(b) $\Rightarrow$ L(a) <m L(b) /* Load$\rightarrow$Load */
  - Si L(a) <p S(b) $\Rightarrow$ L(a) <m S(b) /* Load$\rightarrow$Store */
  - Si S(a) <p S(b) $\Rightarrow$ S(a) <m S(b) /* Store$\rightarrow$Store */
  - Si S(a) <p L(b) $\Rightarrow$ S(a) <m L(b) /* Store$\rightarrow$Load */
  - Igual para las operaciones atómicas (RMW)

Reglas de ordenación SC

		Op2		
		Load	Store	RMW
Op1	Load	X	X	X
	Store	X	X	X
	RMW	X	X	X

- Modelo empleado en el MIPS R10000

Motivación Total Store Order (TSO)

Motivación Total Store Order (TSO)

¿Qué resultados podemos esperar?

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

- En procesadores x86 como los de Intel y AMD, y también en los sparc de Oracle podemos obtener X=0 e Y=0
- ¿Por qué?

Motivación Total Store Order (TSO)

¿Qué resultados podemos esperar?

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

- En procesadores x86 como los de Intel y AMD, y también en los sparc de Oracle podemos obtener X=0 e Y=0
- ¿Por qué?
- ¡Es necesario reordenar código para optimizar el rendimiento!
- El procesador no se puede quedar bloqueado en las escrituras
- Uso del buffer de escritura

El búfer de escritura

- Una escritura necesita permisos para escribir
- Por ejemplo, el protocolo de coherencia puede tener que invalidar las otras copias antes de obtener el permiso de escritura
- Esta operación es muy costosa
- Solución
  1. Realizar la escritura en el búfer de escritura
  2. Continuar con la ejecución del programa
  3. Cuando tengamos permisos de escritura, escribir en la caché
- ¿Cómo se ejecutaría este código asumiendo un búfer de escritura? $\Rightarrow$ Puede dar X=0, Y=0

TOTAL STORE ORDER (TSO)

- Formalmente:
  - Si L(a) <p L(b) $\Rightarrow$ L(a) <m L(b) /* Load$\rightarrow$Load */
  - Si L(a) <p S(b) $\Rightarrow$ L(a) <m S(b) /* Load$\rightarrow$Store */
  - Si S(a) <p S(b) $\Rightarrow$ S(a) <m S(b) /* Store$\rightarrow$Store */
  - /* No Store$\rightarrow$Load! */
  - Orden en las operaciones atómicas (RMW)

Total Store Order (TSO)

- Formalmente:
  - Si L(a) <p L(b) ⇒ L(a) <m L(b) /* Load→Load */
  - Si L(a) <p S(b) ⇒ L(a) <m S(b) /* Load→Store */
  - Si S(a) <p S(b) ⇒ S(a) <m S(b) /* Store→Store */
  - /* No Store→Load! */
  - Orden en las operaciones atómicas (RMW)

Reglas de ordenación TSO

		Op2		
		Load	Store	RMW
Op1	Load	X	X	X
	Store	**B**	X	X
	RMW	X	X	X

- **B** quiere decir que en caso de que una lectura encuente la misma dirección en el búfer de escritura, se tomará de ahí, y no de la última escritura a memoria

FENCES

- Una *fence* (o barrera de memoria) se usa para evitar reordenamientos a través de ella
- Nunca un par de accesos, estando cada uno a un lado de la *fence* se pueden reordenar

FENCES

- Una *fence* (o barrera de memoria) se usa para evitar reordenamientos a través de ella
- Nunca un par de accesos, estando cada uno a un lado de la *fence* se pueden reordenar

REGLAS DE ORDENACIÓN TSO CON FENCES

		Op2			
		Load	Store	RMW	Fence
Op1	Load	X	X	X	**X**
	Store	**B**	X	X	**X**
	RMW	X	X	X	**X**
	Fence	**X**	**X**	**X**	**X**

Fences

- Un código que tenga *fences* entre todos los acesos a memoria se comporta siempre como SC bajo cualquier modelo de memoria

Código que se comporta como SC bajo TSO

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

¿Por qué TSO?

- ¿Es TSO un buen modelo si permite comportamientos extraños?

¿Por qué TSO?

- ¿Es TSO un buen modelo si permite comportamientos extraños?
- Si, es bastante bueno, y funciona para la mayoría de los códigos
- Por ejemplo:

Sincronización con flags

```
/* Initially X = flag = 0 */

X = 1;              |   while (flag == 0);
flag = 1;           |   $r1 = X;
```

Motivación Weak Consistency (WC)[2]

Sincronización con flags

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

X = 1;          |  while (flag == 0);
Y = 2;          |  $r1 = X;
flag = 1;       |  $r2 = Y;
```

- ¿Necesitamos mantener el orden de las dos lecturas y de las dos escrituras a X e Y?

[2] M. Dubois *et al.*, "Memory Access Buffering in Multiprocessors", ISCA, June 1986.

Motivación Weak Consistency (WC)[2]

Sincronización con flags

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

X = 1;                  | while (flag == 0);
Y = 2;                  | $r1 = X;
flag = 1;               | $r2 = Y;
```

- ¿Necesitamos mantener el orden de las dos lecturas y de las dos escrituras a X e Y?
- No, no hace falta. Da igual que se ejecute primero una o la otra
- Sólo hace falta mantener el orden cuando queramos sincronizar los hilos

[2] M. Dubois *et al.*, "Memory Access Buffering in Multiprocessors", ISCA, June 1986.

Weak Consistency (WC)

- La syncronización actua como fences
- Sequential Consistency for Data-Race-Free programs (SC for DRF)[3]
- La syncronización debe ser expuesta al hardware

[3] S. V. Adve and M. D. Hill. “Weak Ordering–A New Definition”, ISCA, May 1990

Weak Consistency (WC)

- La syncronización actua como fences
- Sequential Consistency for Data-Race-Free programs (SC for DRF)[3]
- La syncronización debe ser expuesta al hardware

Reglas de ordenación WC

		Op2			
		Load	Store	RMW	**Sync**
Op1	Load	**A**	**A**	**A**	X
	Store	B	**A**	**A**	X
	RMW	**A**	**A**	**A**	X
	Sync	X	X	X	X

[3] S. V. Adve and M. D. Hill. “Weak Ordering–A New Definition”, ISCA, May 1990

Motivación Release Consistency (RC)[4]

Sincronización con flags

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

X = 1;          |  while (flag == 0);
Y = 2;          |  $r1 = X;
flag = 1;       |  $r2 = Y;
```

- ¿Realmente necesitamos mantener siempre el orden en la sincronización?

[4] K. Gharachorloo *et al.*, "Memory Consistency and Event Ordering in Scalable Shared-Memory", ISCA, May 1990.

Motivación Release Consistency (RC)[4]

Sincronización con flags

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

X = 1;              |  while (flag == 0);
Y = 2;              |  $r1 = X;
flag = 1;           |  $r2 = Y;
```

- ¿Realmente necesitamos mantener siempre el orden en la sincronización?
- No, no todos los órdenes.
- Se pueden definir dos tipos de sincronización: Acquire y **Release**
  - Flag = 1 es una operación con semánticas Release y el bucle while es una operación con semánticas Acquire

[4] K. Gharachorloo *et al.*, "Memory Consistency and Event Ordering in Scalable Shared-Memory", ISCA, May 1990.

Release Consistency (RC)

- Hay que asegurar solo ACQ→Load,Store y Load,Store→REL
- Acquire y Release expuestos al hardware
- Sequential Consistency for Data-Race-Free programs (SC for DRF)

Release Consistency (RC)

- Hay que asegurar solo ACQ→Load,Store y Load,Store→REL
- Acquire y Release expuestos al hardware
- Sequential Consistency for Data-Race-Free programs (SC for DRF)

Reglas de ordenación RC

		Op2				
		Load	Store	RMW	**ACQ**	**REL**
Op1	Load	A	A	A	**A**	X
	Store	B	A	A	**A**	X
	RMW	A	A	A	**A**	X
	ACQ	X	X	X	X	X
	REL	**A**	**A**	**A**	X	X

Outline

Motivación

- La consistencia de memoria normalmente usa el protocolo de coherencia de cache como una “caja negra”
- Esta caja no romple el modelo de consistencia de la máquina
  - Garantiza SC para un sistema que no reordena accesos
- Esto se hace porque es más facil razonar sobre consistencia y coherencia por separado
- Sin embargo, la mayoría de los procesadores actuales no garantizan SC
- Además, lenguajes como C++ o Java han adoptado el modelo SC for DRF
- El protocolo de coherencia se puede optimizar bastante si tiene en cuenta la consistencia asumida por el procesador o por el lenguaje

PRIMERAS PROPUESTAS: SOFTWARE COHERENCE

- Varias propuestas a finales de los 80
- Basadas en auto-invalidación
  - El compilador inserta instrucciones de auto-invalidacion
  - La instrucción invalida los bloques en las cachés privadas
- Garantizan Weak Consistency
  - Insertan la auto-invalidación al final de bucles FOR-ALL

PRIMERAS PROPUESTAS: SOFTWARE COHERENCE

- Lee *et al.*[5] proponen que el compilador marque instrucciones no cacheables e instrucciones cacheables y auto-invalidables
- Hay que hacer la auto-invalidación de forma selectiva
- Cheong *et al.*[6] proponen que el compilador marque las lecturas como *memory-load* o *cache-load*. Usa un bit *clean* por línea de cache. Cuando llega una sincronización los bits *clean* se ponen a falso. Memory-loads a bloques con el bit a falso se consideran fallos
  - Combinación del análisis del compilador y del comportamiento en tiempo de ejecución
- Ambas propuestas usan write-through

[5] R. L. Lee *et al.*, "Multiprocessor Cache Design Considerations", ISCA, June 1987.
[6] H. Cheong and A. V. Vaidenbaum, "A Cache Coherece Scheme with Fast Selective Invalidation", ISCA, June 1988.

Dynamic self-invalidation (DSI)

- Dynamic self-invalidation propone el uso de auto-invalidación para reducir el tiempo de las invalidaciones debido a las escrituras
- El directorio predice qué bloques hay que auto-invalidar y cuándo
- Proponen dos modos de funcionamiento
  - Sequential Consistency (SC)
    - Requiere invalidaciones ante una escritura (en caso de que no se hayan auto-invalidado)
    - Las auto-invalidaciones informan al directorio
  - Weak Consistency (WC)
    - Uso de copias *tear-off*, que no se mantienen en el directorio
    - Las auto-invalidaciones se hacen en cada punto de sincronización y no informan al directorio (se reduce el tráfico)
- Downgrades realizados ante una lectura

[7] A. R. Lebeck and D. A. Wood, “Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors”, ISCA, June 1995.

DeNovo

- Auto-invalidación $\Rightarrow$ No necesita información de compartidores
  - Insertada por el programador o el modelo (disciplined programming)
  - Touched bits para filtrar invalidaciones
- Downgrades mediante lectura $\Rightarrow$ Necesita información del owner
- Deja la sincronización como trabajo futuro

[8] B. Choi *et al.*, "DeNovo: Rethinking the memory hierarchy for disciplined parallelism", PACT, Oct. 2011.

Auto-invalidación selectiva con Bloom filters[9]

- Self-donwgrade de todos los bloques en cache (o write-through)
  - Introduce la posibilidad de multiples escritores
  - No se puede hacer una escritura en memoria del bloque entero
    - Se escribe solo lo modificado (diffs)
  - Hay que mantener la pista de lo modificado en caché
    - Un bit por palabra en caché (sobrecarga de 12.5%)
- Auto-invalidación selectiva con Bloom filters
  - Los procesadores marcan las escrituras en el filtro
  - Ante un Release, escriben el contenido del filtro en una posición de memoria (asociada al cerrojo que hace el release)
  - Cuando un lock hace un acquire lee el filtro y cuando hace el release lo junta con el que tiene de sus escrituras
  - Los filtros se resetean en las barreras
- Sincronización: No cacheable, acceso a cache compartida

[9] T. J. Ashby *et al.*, "Software-based cache coherence with hardware-assisted selective self-invalidations using bloom filters", IEEE Transactions on Computers, April 2011.

VIPS-M[10,11,12,13,14]

- Auto-invalidación y self-downgrade selectivo
  - Basados en una clasificación de privado/compartido por el OS
- Provee RC, así que también SC for DRF
- Ideal para usar cachés virtualmente etiquetadas[11]
- Ideal para jerarquías de cache parcialmente compartidas[12]
- Sincronización eficiente[13]
- Ideal para distributed shared memory[14]

[10] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence", PACT, 2012.
[11] S. Kaxiras and A. Ros, "A new perspective for efficient virtual-cache coherence", ISCA, 2013.
[12] A. Ros *et al.*, "Hierarchical Private/Shared Classification: the Key to Simple and Efficient Coherence for Clustered Cache Hierarchies", HPCA, 2015.
[13] A. Ros and S. Kaxiras, "Callback: Efficient Synchronization without Invalidation with a Directory Just for Spin-Waiting", ISCA, 2015.
[14] S. Kaxiras *et al.*, "Turning Centralized Coherence and Distributed Critical-Section Execution on their Head: A New Approach for Scalable Distributed Shared Memory", HPDC, 2015.

TSO-CC[15] y RC3[16]

- TSO-CC provee TSO, más estricto que *SC for DRF*
  - Más fuerte y usado por la mayoría de los procesadores
- Load→Load: auto-invalida toda la cache ante un fallo
  - Solo si el valor es reciente (timestamps)
  - Para evitar que la espera ocupada dure siempre las lecturas solo pueden acertar en cache un número determinado de veces
- Store→Store: realiza las escrituras en memoria en orden
- Load→Store: las escrituras no se retiran del buffer de escritura hasta que todas las lecturas previas se han completado
- RC3: usa el conocimiento de sincronización, si existe, para optimizar TSO-CC

[15] M. Elver and V. Nagarajan, "TSO-CC: Consistency directed cache coherence for TSO" HPCA, 2014.
[16] M. Elver and V. Nagarajan, "RC3: Consistency directed cache coherence for x86-64 with RC extensions" PACT, 2015.

SPEL[17,18]

- Protocolo de consistecia dual
- Modo SC: para código que no es DRF
  - Protocolo MOESI
- Modo WC: para código que no DRF
  - Protocolo basado en auto-invalidación
- Diferencia entre regiones usando el modelo de programación (OMP)
- Transición entre los dos modos bajo demanda

[17] A. Ros and A. Jimborean, “A Dual-Consistency Cache Coherence Protocol”, IPDPS, May 2015.
[18] A. Ros and A. Jimborean, “A Hybrid Static-Dynamic Classification for Dual-Consistency Cache Coherence”, TPDS, Mar. 2016.

Outline

Motivation

Cited from [Sorin, Hill, and Wood 2010]

"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. ..."

- To satisfy this definition protocols react to writes and invalidate all cached copies
- This is a source of significant complexity and cost
  - Snooping, broadcast, or a directory to track copies
    - → Power, invalidation, and area overhead
  - Additional states for performance (e.g., Exclusive, Owner)
    - → Explosion in the number of transient states

Motivation

- Directory/Snooping Coherence is a relic of NUMA/SMP systems of the past
  - Does not take into account the on-chip memory hierarchy (e.g., Owned state for $-to-$ transfers)
    - → Shared LLC cache
- MESI is already complex
  - Although we know how to verify it, any optimization needs verification from the beginning
- A new class of manycores is now appearing:
  - Many very simple cores
    - → Coherence overhead is a big issue
  - Coherence is sporadically needed
    - → Why pay always?
- Our goal → Simplify coherence

Motivation

- Write-through protocols are simple
  - Only Valid and Invalid states

Motivation

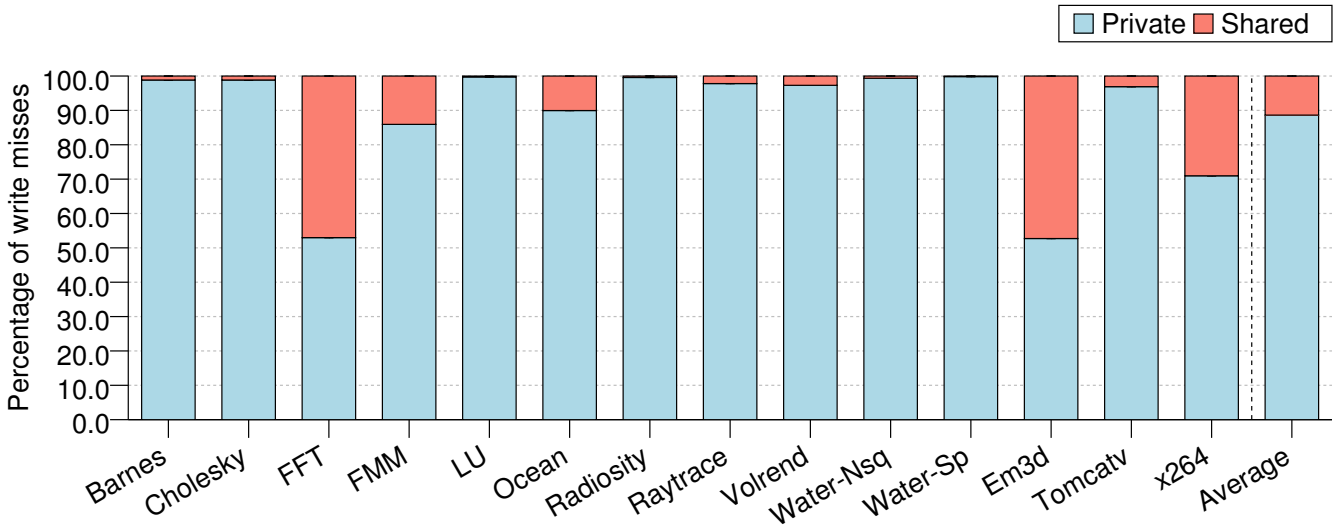
- Write-through protocols are simple
  - Only Valid and Invalid states
  - But they are not efficient because of write misses
- Which write misses?

Motivation

- Write-through protocols are simple
  - 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

Motivation

- Write-through protocols are simple
  - 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%



SIMPLIFYING COHERENCE: WRITE POLICY

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

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

PRIVATE/SHARED CLASSIFICATION

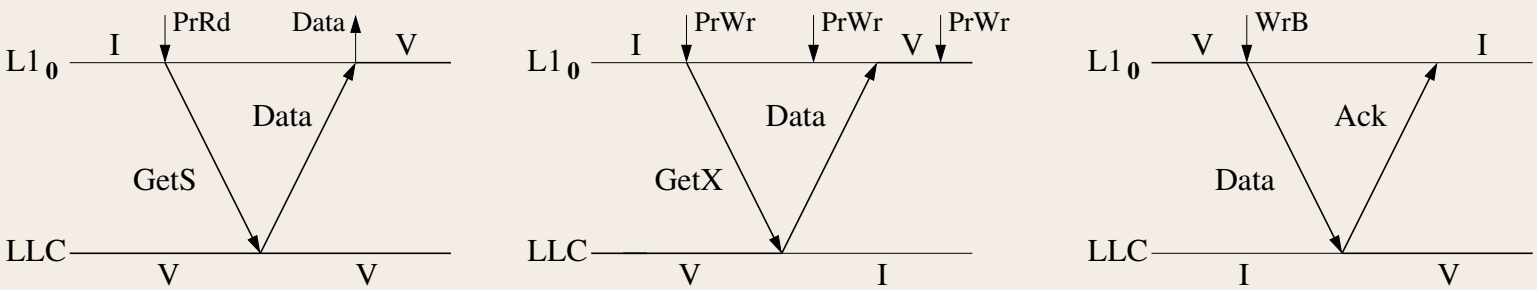
- Classify data (cache blocks) into private and shared
  - A-priori: Before issuing the coherence transaction we know if it is for a private or for a shared block
    - i.e., OS/TLB, compiler, application
    - Private and shared protocols completely independent
    - Can be verified separately
- Page-level classification using the OS and the TLBs
  - Both page table and TLB entries have a P/S bit
  - The first TLB miss by a core sets the page to P
  - Subsequent TLB misses set the page to S
    - Interrupts the single core having the page as P
    - Forces the WT of every dirty block in the page

DELAYED WRITE-THROUGHS

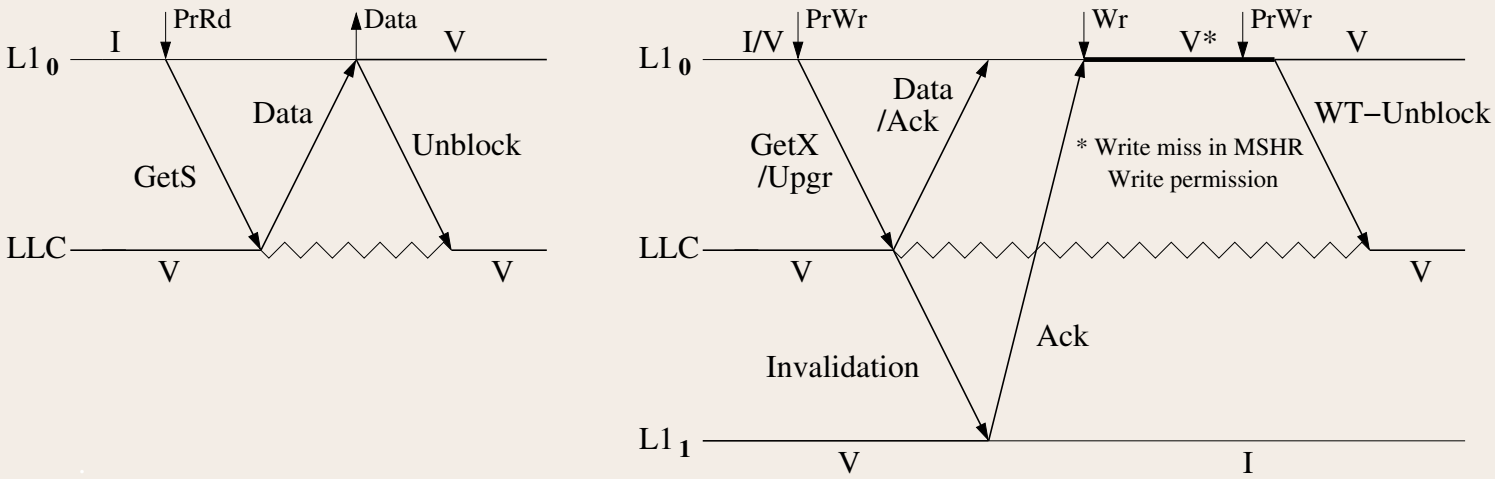
- We still have some (10%) write misses for shared blocks
- The obvious optimization to a WT policy is to delay WTs to coalesce as many writes as possible on the same cache line for performance
- Implementation on MSHRs
  - We add a hidden transient state (no races)
  - WTs triggered by timers (after 500-1000 cycles) or because of lack of MSHR capacity

TWO INDEPENDENT PROTOCOLS

READ, WRITE, AND WRITE-BACK TRANSACTIONS FOR **PRIVATE** BLOCKS



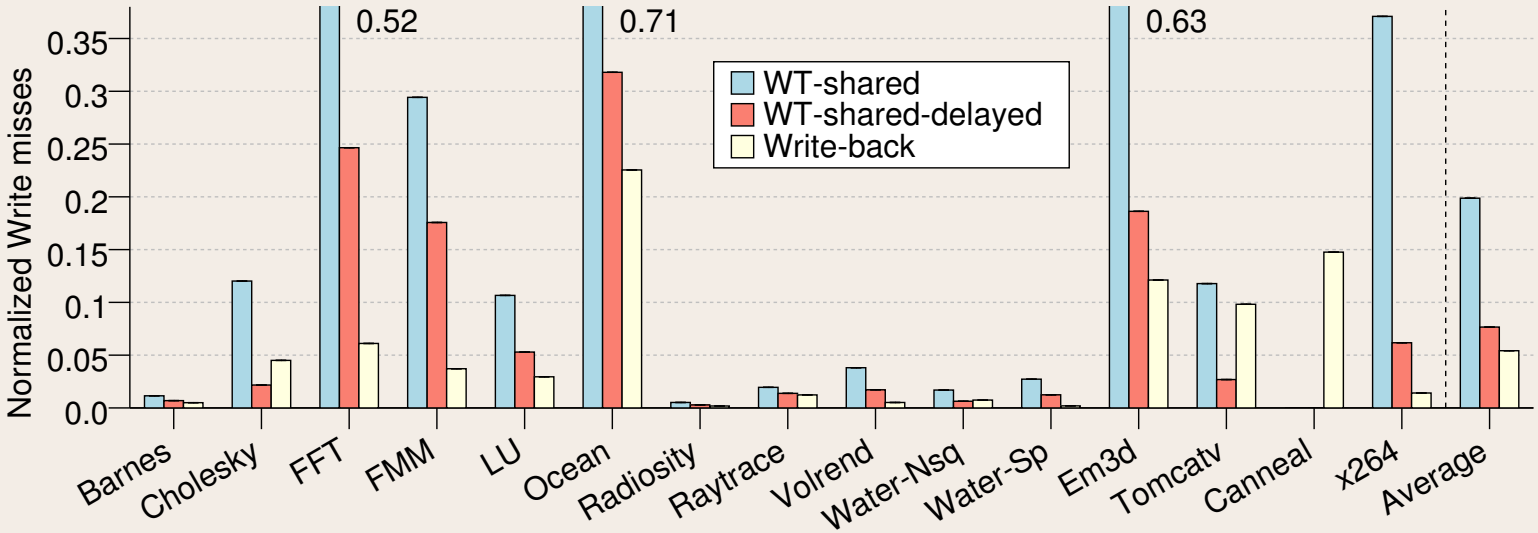
READ AND WRITE TRANSACTIONS FOR **SHARED** BLOCKS



WRITE MISSES

- Write misses normalized w.r.t. a simple WT policy (not shown)
- 80% reduction by using write-back policy for private blocks
- 92.5% reduction when delayed WT are employed for shared blocks

REDUCTION IN WRITE MISSES DUE TO P/S CLASSIF. AND DELAYED WT



VIPS CONCLUSIONS

- Simplifies the protocol to just two states (VI)
- Write-throughs eliminate the need of tracking writers at the directory
  - → Area reduction
- No indirection for read misses
  - → Correct shared data always at the LLC
- Supports sequential consistency for every application
  - Same consistency model as the more complex MESI
- But we still have invalidations and directory blocking...

MOTIVATION: CONSISTENCY

- Coherence must be “invisible” to the memory consistency model
  - → It must allow sequential consistency (SC) in the presence of data races
- But what if we only need SC for Data-Race-Free (DRF) programs? *[Adve and Hill, ISCA’99]*
  - Relaxed consistency offers an opportunity to simplify coherence *[SARC Coherence (IEEE Micro’10), DeNovo (PACT’11)]*

SELF-INVALIDATION

- Self-Invalidation of shared data from L1s
  - Selective Flush (SF) upon synchronization points
  - Self-Invalidation can be optimized to exclude read-only data and can be enhanced with speculation *[Speculative Cache Lookup/Coherence Decoupling, Huh et al., ASPLOS'04]*
- We eliminate invalidations
  - The directory is gone!
  - We only offer SC for Data-Race-Free (DRF) programs

Multiple Writers & Merge

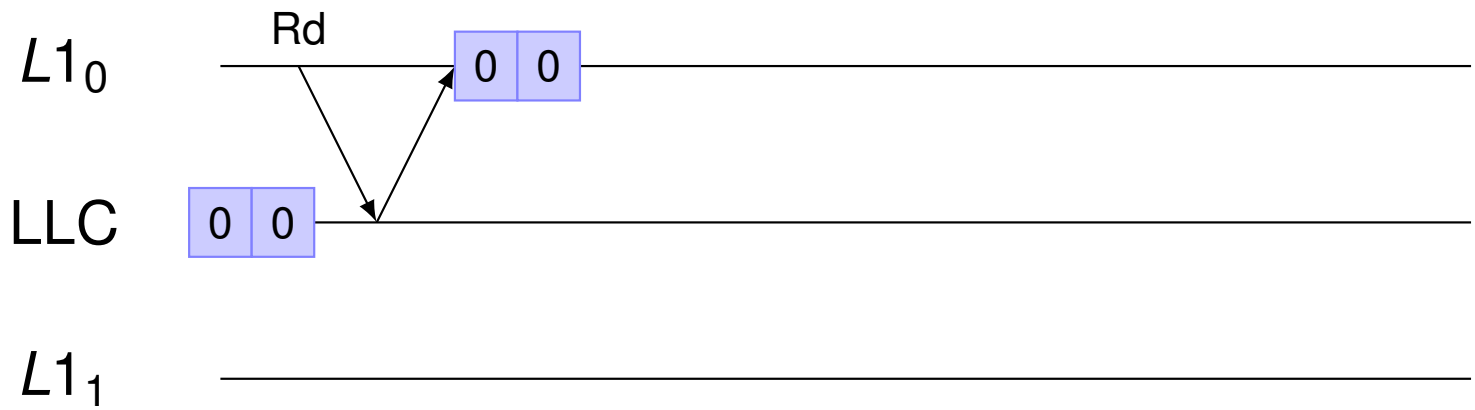
- Self-invalidation offers SC only for programs that are DRF at cache-line granularity
  - But not for DRF at word granularity
  - Does not work with false sharing
    - False sharing on a cache line while DRF at word granularity
    - We do not have invalidations on writes!

Multiple Writers & Merge

- Self-invalidation offers SC only for programs that are DRF at cache-line granularity
  - But not for DRF at word granularity
  - Does not work with false sharing
    - False sharing on a cache line while DRF at word granularity
    - We do not have invalidations on writes!

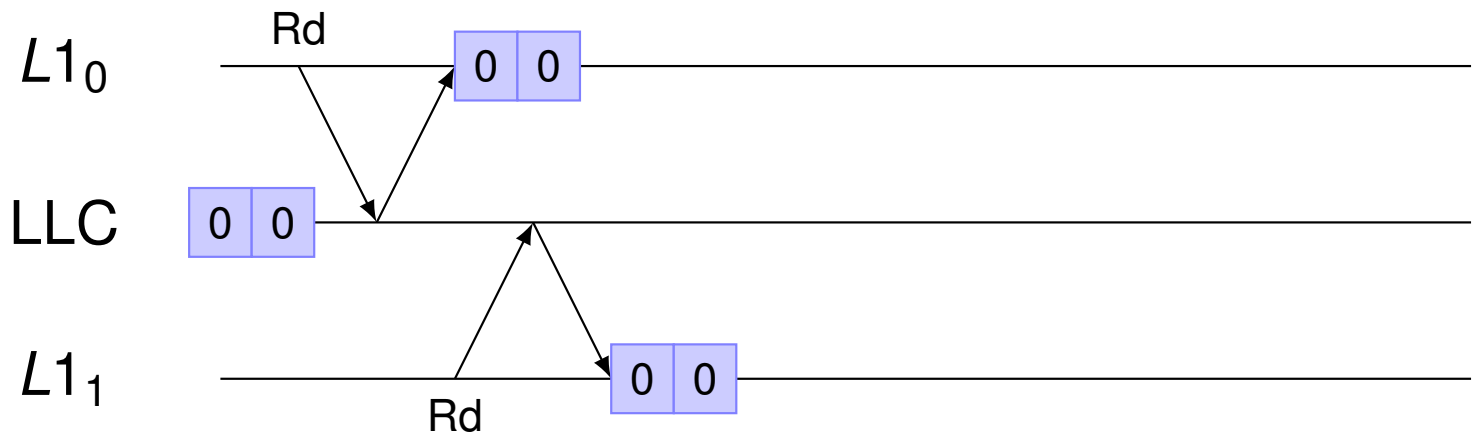
Multiple Writers & Merge

- Self-invalidation offers SC only for programs that are DRF at cache-line granularity
  - But not for DRF at word granularity
  - Does not work with false sharing
    - False sharing on a cache line while DRF at word granularity
    - We do not have invalidations on writes!



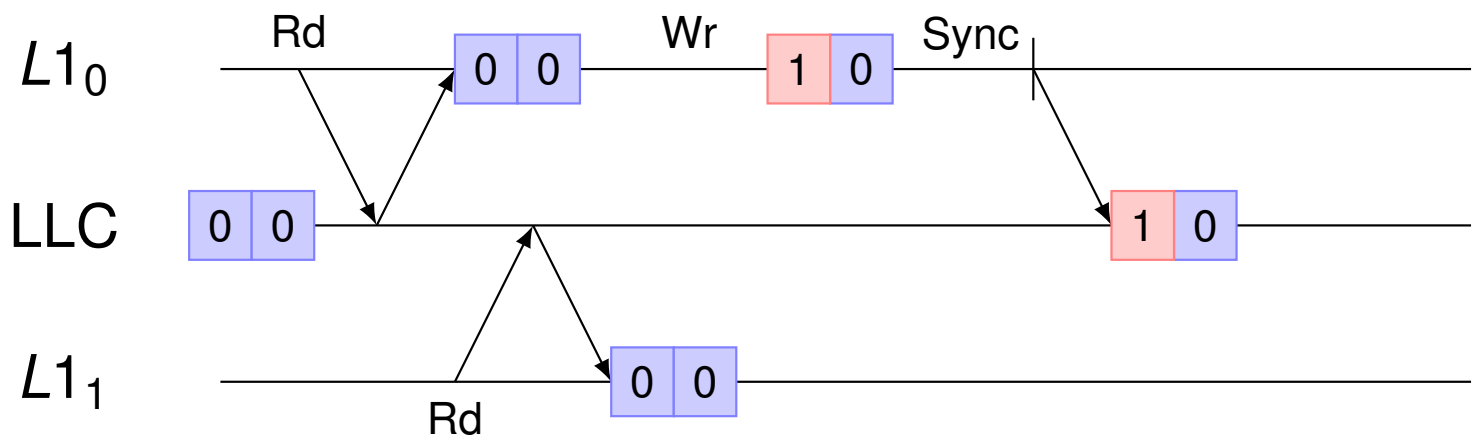
Multiple Writers & Merge

- Self-invalidation offers SC only for programs that are DRF at cache-line granularity
  - But not for DRF at word granularity
  - Does not work with false sharing
    - False sharing on a cache line while DRF at word granularity
    - We do not have invalidations on writes!



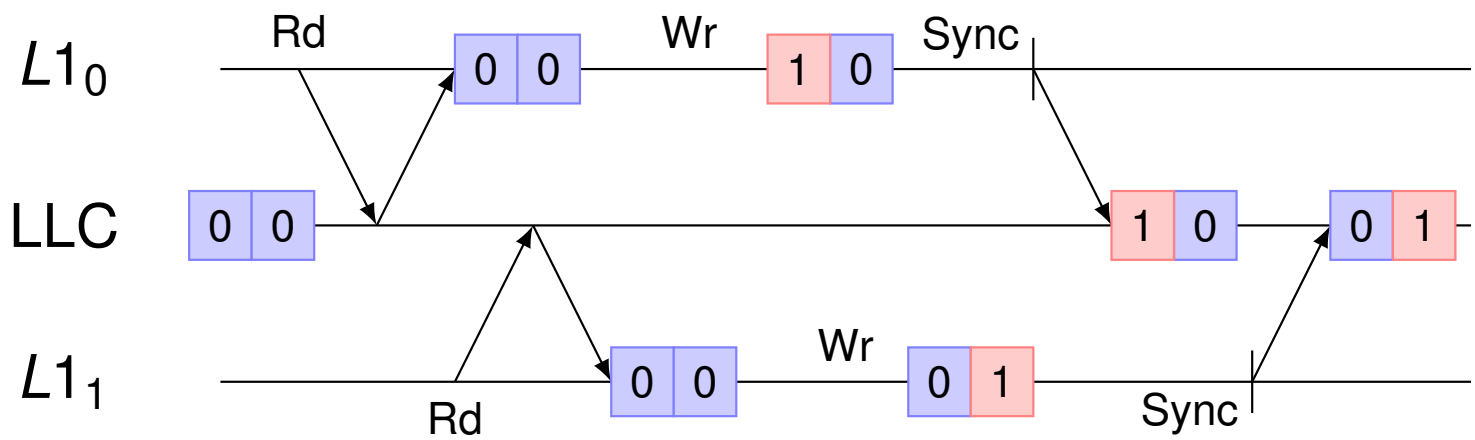
Multiple Writers & Merge

- Self-invalidation offers SC only for programs that are DRF at cache-line granularity
  - But not for DRF at word granularity
  - Does not work with false sharing
    - False sharing on a cache line while DRF at word granularity
    - We do not have invalidations on writes!



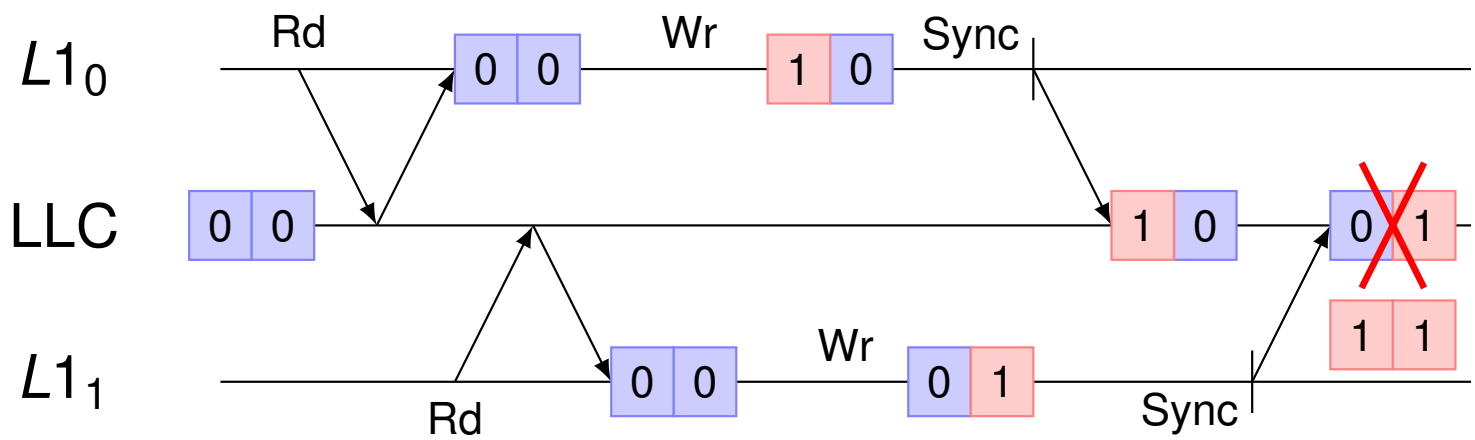
Multiple Writers & Merge

- Self-invalidation offers SC only for programs that are DRF at cache-line granularity
  - But not for DRF at word granularity
  - Does not work with false sharing
    - False sharing on a cache line while DRF at word granularity
    - We do not have invalidations on writes!



Multiple Writers & Merge

- Self-invalidation offers SC only for programs that are DRF at cache-line granularity
  - But not for DRF at word granularity
  - Does not work with false sharing
    - False sharing on a cache line while DRF at word granularity
    - We do not have invalidations on writes!

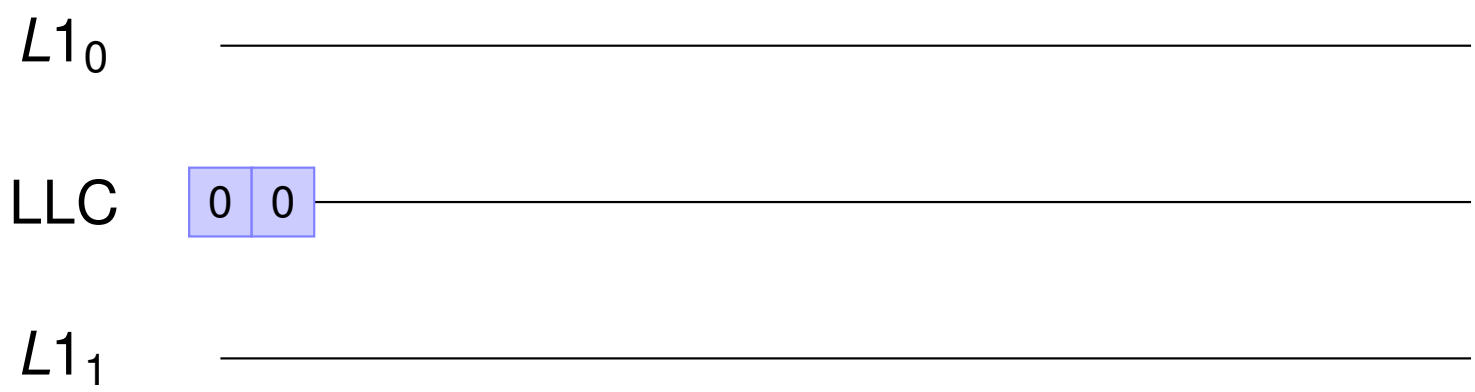


Multiple Writers & Merge

- This is solved by writing through only the modified words (send cache-line DIFFs) and merging them in the LLC
  - Cache structures are not modified
  - Written words bits only in MSHRs
    - Our “dirty” state is transient!

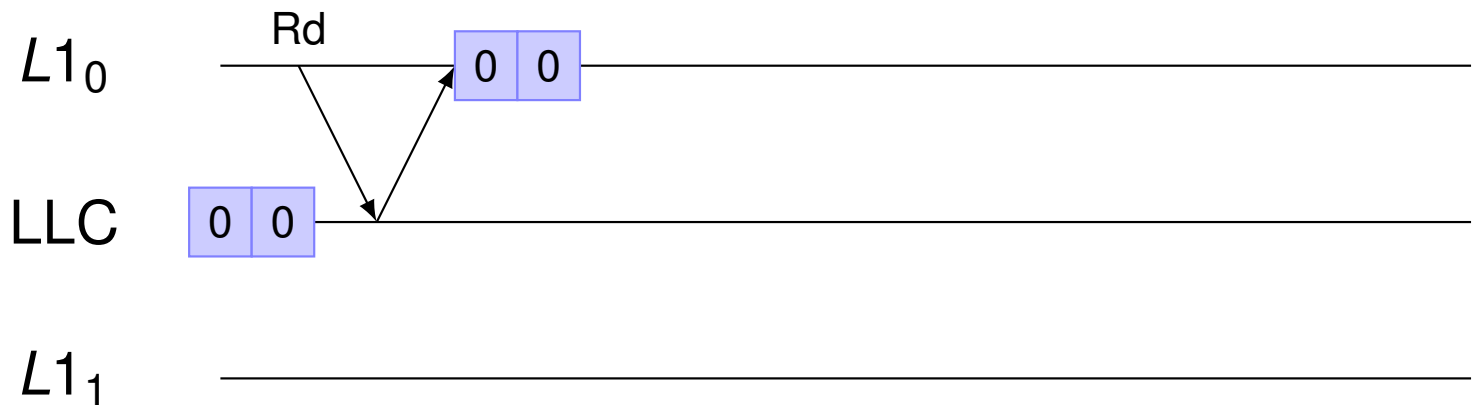
Multiple Writers & Merge

- This is solved by writing through only the modified words (send cache-line DIFFs) and merging them in the LLC
  - Cache structures are not modified
  - Written words bits only in MSHRs
    - Our “dirty” state is transient!



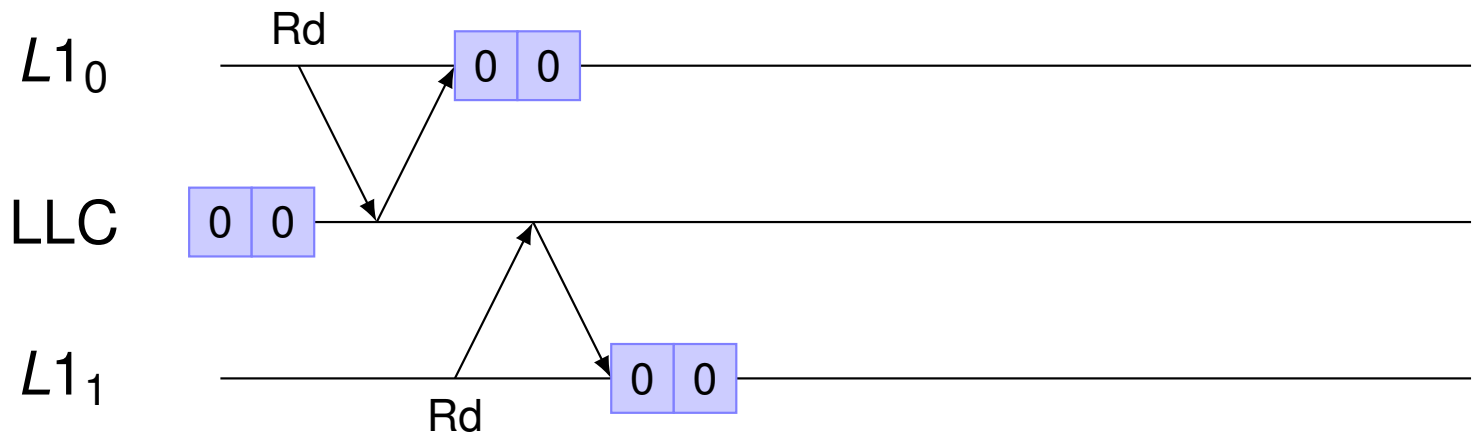
Multiple Writers & Merge

- This is solved by writing through only the modified words (send cache-line DIFFs) and merging them in the LLC
  - Cache structures are not modified
  - Written words bits only in MSHRs
    - Our “dirty” state is transient!



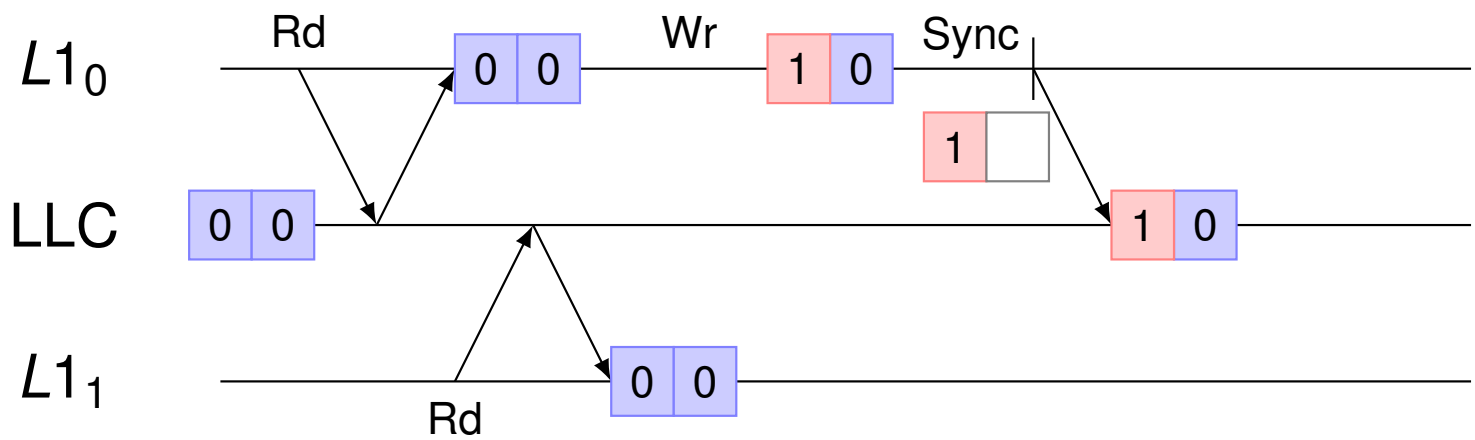
Multiple Writers & Merge

- This is solved by writing through only the modified words (send cache-line DIFFs) and merging them in the LLC
  - Cache structures are not modified
  - Written words bits only in MSHRs
    - Our “dirty” state is transient!



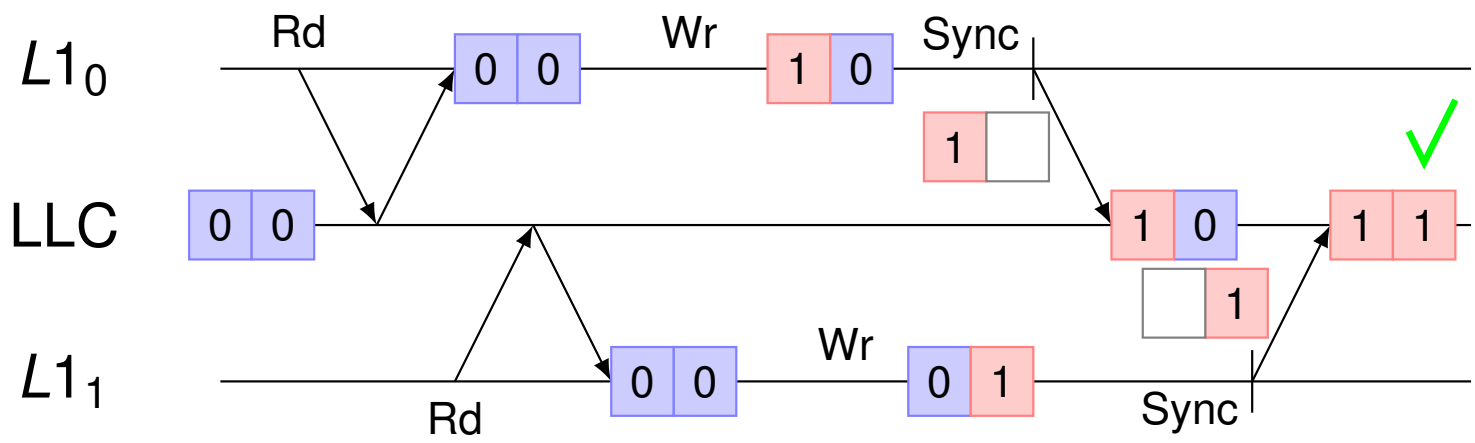
Multiple Writers & Merge

- This is solved by writing through only the modified words (send cache-line DIFFs) and merging them in the LLC
  - Cache structures are not modified
  - Written words bits only in MSHRs
    - Our “dirty” state is transient!



Multiple Writers & Merge

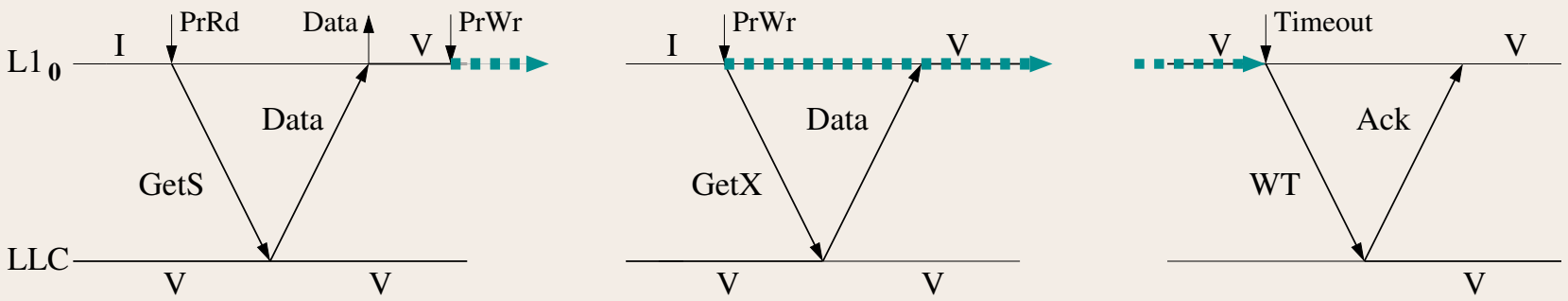
- This is solved by writing through only the modified words (send cache-line DIFFs) and merging them in the LLC
  - Cache structures are not modified
  - Written words bits only in MSHRs
    - Our “dirty” state is transient!



PROTOCOL FOR DRF BLOCKS

- Simple request-response protocol for DRF blocks!

READ, WRITE, AND DELAYED WRITE-THROUGH TRANSACTIONS



SYNCHRONIZATION

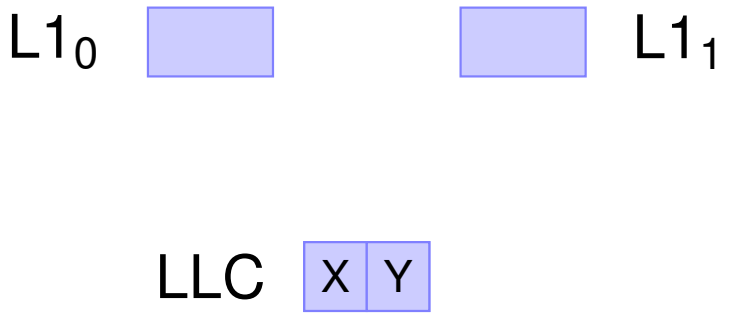
- Our simple protocol works for private and shared (DRF) blocks
- What about synchronization? It relies on data races!
  - Readers never “see” a new write (unless they get flushed from the cache)
- A synchronization data-race must “see” the new writes
  - Solution: Do not create an L1 copy for synchronization instructions (atomic RMW)
    - → Go directly to the LLC where the write throughs happen
  - The LLC line remains blocked during atomic transactions
    - Delayed WTs eliminate spinning for short critical sections [details in the paper]

EXAMPLE OF EXECUTION

EXAMPLE OF DRF CODE

```
/* Initially X & Y = 0 */
/* X & Y in the same block */

X = 1;                    | $r2 = Y;
SIGNAL(cond);             | WAIT(cond);
                          | $r1 = X;
```



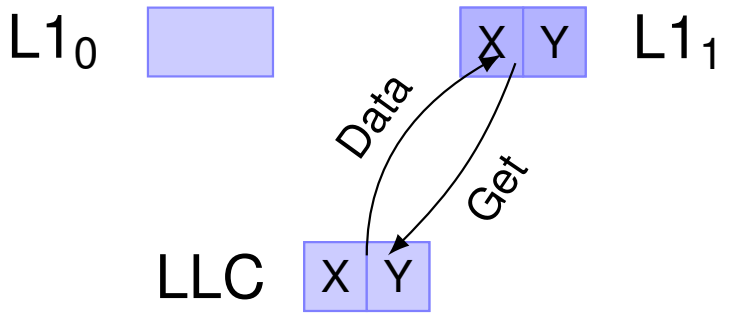
- SIGNAL has release semantics
- WAIT has aquire semantics
- Aquire and release semantics guarantee that read returns the value generated by the write
  - SIGNAL $\Rightarrow$ self-downgrade, WAIT $\Rightarrow$ self-invalidate
- Implemented using atomic operations (not shown)

EXAMPLE OF EXECUTION

EXAMPLE OF DRF CODE

```
/* Initially X & Y = 0 */
/* X & Y in the same block */

X = 1;                    | $r2 = Y;
SIGNAL(cond);             | WAIT(cond);
                          | $r1 = X;
```



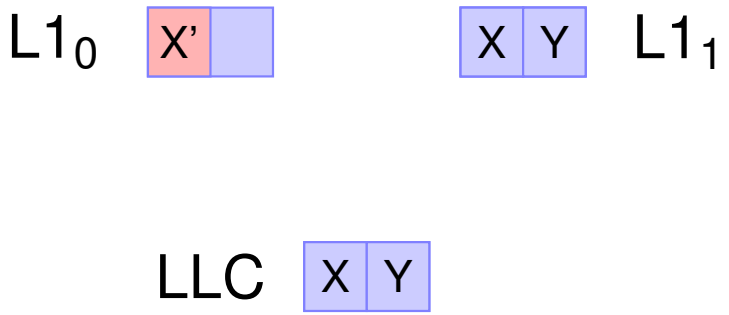
- SIGNAL has release semantics
- WAIT has aquire semantics
- Aquire and release semantics guarantee that read returns the value generated by the write
  - SIGNAL $\Rightarrow$ self-downgrade, WAIT $\Rightarrow$ self-invalidate
- Implemented using atomic operations (not shown)

EXAMPLE OF EXECUTION

EXAMPLE OF DRF CODE

```
/* Initially X & Y = 0 */
/* X & Y in the same block */

X = 1;                 |  $r2 = Y;
SIGNAL(cond);          |  WAIT(cond);
                       |  $r1 = X;
```



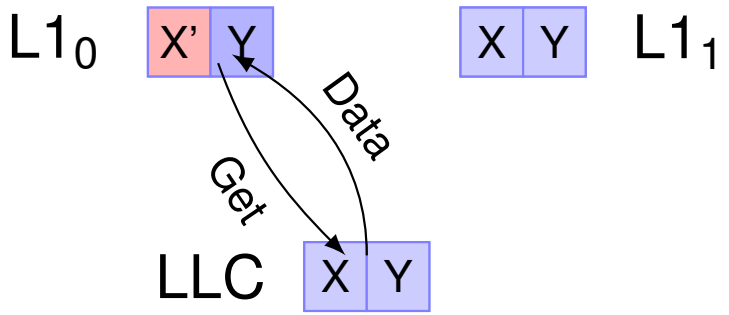
- SIGNAL has release semantics
- WAIT has aquire semantics
- Aquire and release semantics guarantee that read returns the value generated by the write
  - SIGNAL $\Rightarrow$ self-downgrade, WAIT $\Rightarrow$ self-invalidate
- Implemented using atomic operations (not shown)

EXAMPLE OF EXECUTION

EXAMPLE OF DRF CODE

```
/* Initially X & Y = 0 */
/* X & Y in the same block */

X = 1;                 |  $r2 = Y;
SIGNAL(cond);          |  WAIT(cond);
                       |  $r1 = X;
```



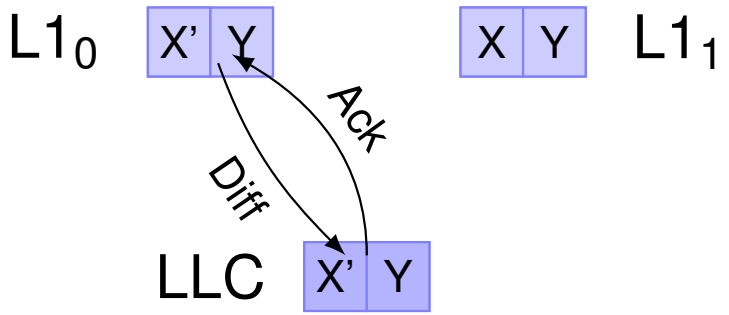
- SIGNAL has release semantics
- WAIT has aquire semantics
- Aquire and release semantics guarantee that read returns the value generated by the write
  - SIGNAL $\Rightarrow$ self-downgrade, WAIT $\Rightarrow$ self-invalidate
- Implemented using atomic operations (not shown)

EXAMPLE OF EXECUTION

EXAMPLE OF DRF CODE

```
/* Initially X & Y = 0 */
/* X & Y in the same block */

X = 1;                  | $r2 = Y;
SIGNAL(cond);           | WAIT(cond);
                        | $r1 = X;
```



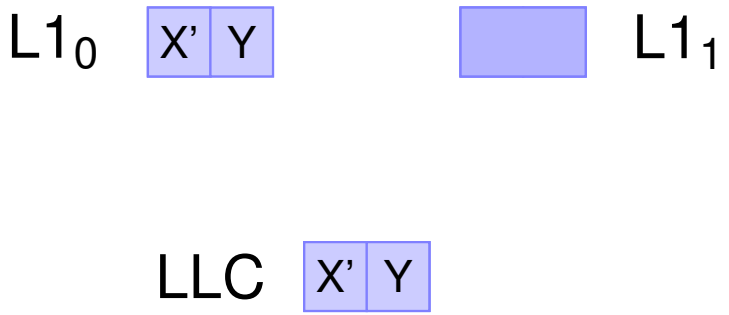
- SIGNAL has release semantics
- WAIT has aquire semantics
- Aquire and release semantics guarantee that read returns the value generated by the write
  - SIGNAL $\Rightarrow$ self-downgrade, WAIT $\Rightarrow$ self-invalidate
- Implemented using atomic operations (not shown)

EXAMPLE OF EXECUTION

EXAMPLE OF DRF CODE

```
/* Initially X & Y = 0 */
/* X & Y in the same block */

X = 1;                    | $r2 = Y;
SIGNAL(cond);             | WAIT(cond);
                          | $r1 = X;
```



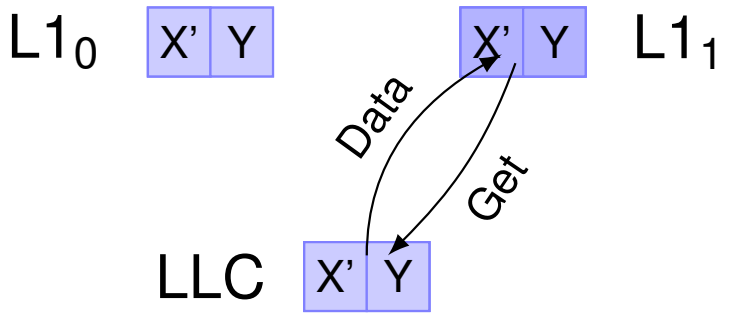
- SIGNAL has release semantics
- WAIT has aquire semantics
- Aquire and release semantics guarantee that read returns the value generated by the write
  - SIGNAL $\Rightarrow$ self-downgrade, WAIT $\Rightarrow$ self-invalidate
- Implemented using atomic operations (not shown)

EXAMPLE OF EXECUTION

EXAMPLE OF DRF CODE

```
/* Initially X & Y = 0 */
/* X & Y in the same block */

X = 1;                  |  $r2 = Y;
SIGNAL(cond);           |  WAIT(cond);
                        |  $r1 = X;
```

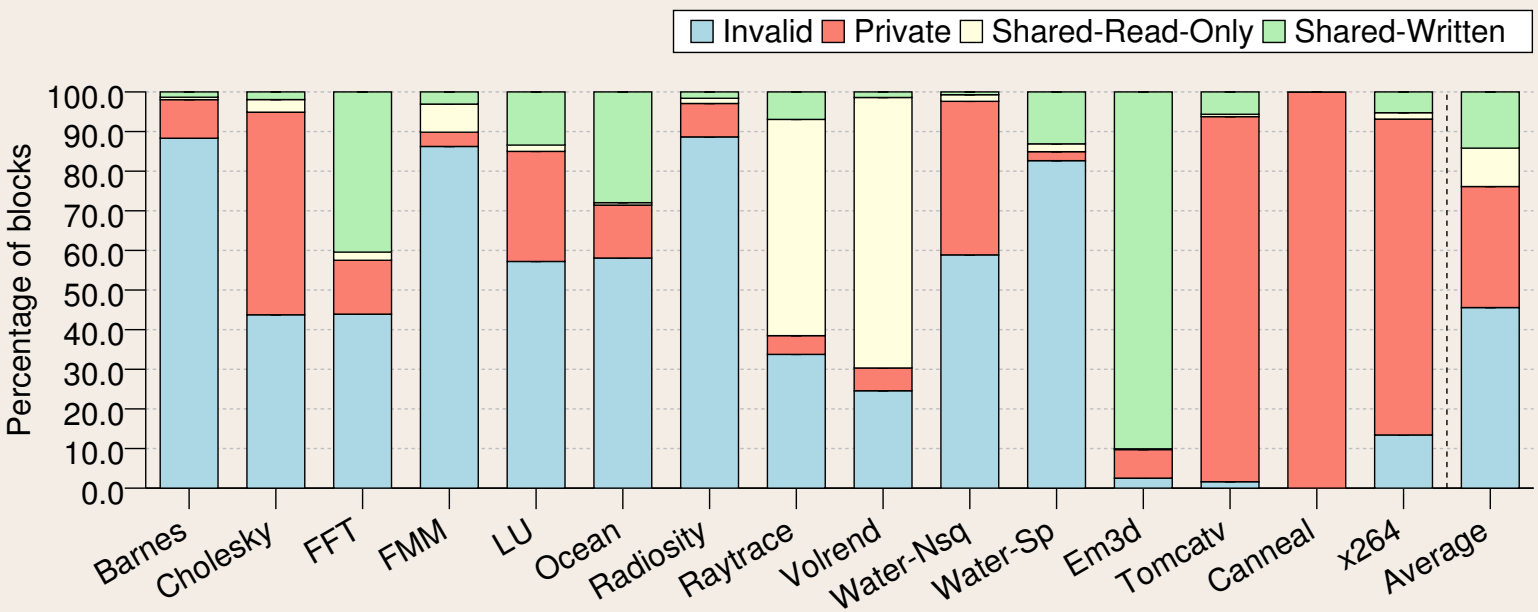


- SIGNAL has release semantics
- WAIT has aquire semantics
- Aquire and release semantics guarantee that read returns the value generated by the write
  - SIGNAL $\Rightarrow$ self-downgrade, WAIT $\Rightarrow$ self-invalidate
- Implemented using atomic operations (not shown)

SELECTIVE FLUSHING

- By only flushing shared-written lines, we prevent about 73.9% of valid cached blocks from being evicted

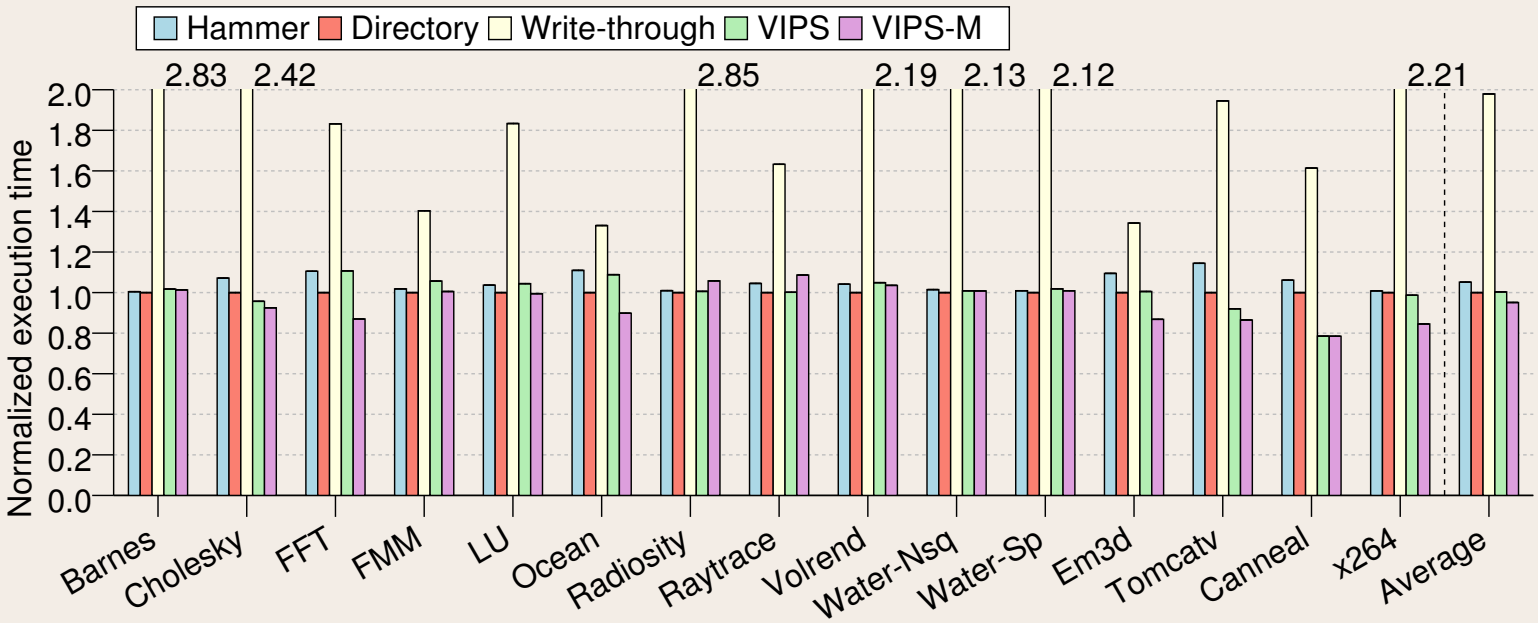
LINES FOUND IN THE CACHE UPON A SELECTIVE FLUSHING



Execution time

- Hammer increases execution time w.r.t. MESI, and the performance of a WT policy is prohibitive
- VIPS performs similar to MESI
- VIPS-M improves MESI by 4.8%, on average

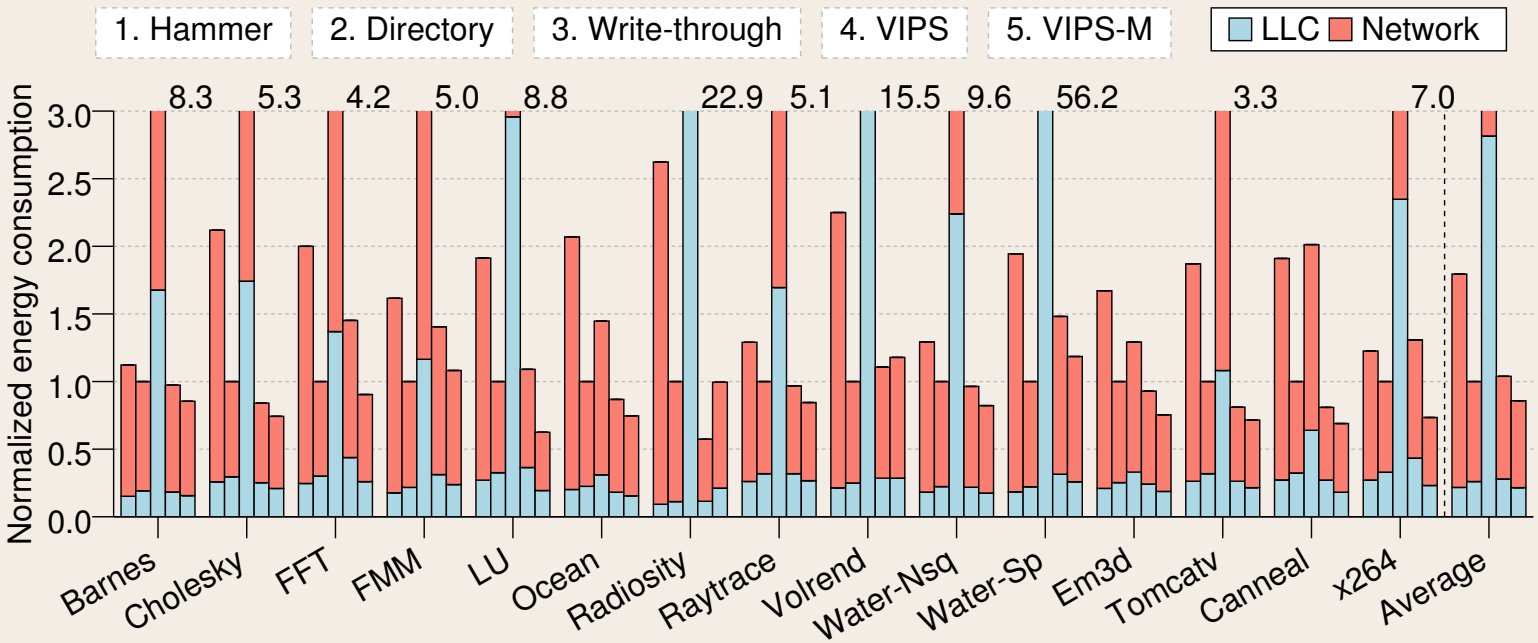
Normalized execution time w.r.t. Directory



Energy consumption

- Hammer and WT consumption is undesirable
- VIPS consumes similar energy to MESI
- VIPS-M reduces consumption by 14.2% mainly due to its lower traffic requirements

Normalized energy consumption w.r.t. Directory



VIPS-M CONCLUSIONS

- Selective flushing eliminates the need to track readers at the directory
  - No need to send invalidations
  - The directory is gone!
- Indirection completely removed
- Private and DRF protocols practically the same
  - → They differ only in when data is written back in the LLC
- Provides correct semantics for synchronization instructions
- Supports sequential consistency for DRF programs

Virtual caches & Cohererence

- Multicore systems usually employ physical caches
  - Coherence is guaranteed per physical block
- Virtual caches have several advantages over physical caches
  - Address translation not in the critical path of cache accesses
  - Performed after the cache miss
    - $\Rightarrow$ Saves TLB energy consumption for cache hits
  - Larger TLBs can be implemented
    - Even unified shared TLBs

Virtual caches & Coherence

- Virtual caches introduce complexity
  - Reverse translation for coherence requests
    - Extra structures and lookups
  - Reverse translation could be avoided by issuing together the physical and the virtual addresses
    - But synonyms (different virtual addresses mapped to the same physical address) prevent this
    - The same physical address can be cached with a different virtual address
- Our claim: VIPS-M can remove virtual cache complexity by eliminating reverse translation even in the face of synonyms

Reverse translation

When is reverse translation required?

- For every coherence message received by a virtual cache that belongs to a transaction initiated by another cache
- In traditional coherence protocols: invalidations, downgrades, and forwardings

Our observation

Virtual-cache coherence without reverse translations is possible with a protocol that does not have invalidations, downgrades, or forwardings, towards the L1s

Reverse translation elimination

VIPS-M eliminates the reverse translation that would be required by virtual caches because:

- It uses self-invalidation
  - ⇒ No invalidations are issued from the LLC to the L1s
- It uses self-downgrade (write-throughs)
  - ⇒ No downgrades are issued from the LLC to the L1s
- It is a request-response protocol (no $-to-$ transfers)
  - ⇒ No forwardings are issued from the LLC to the L1s

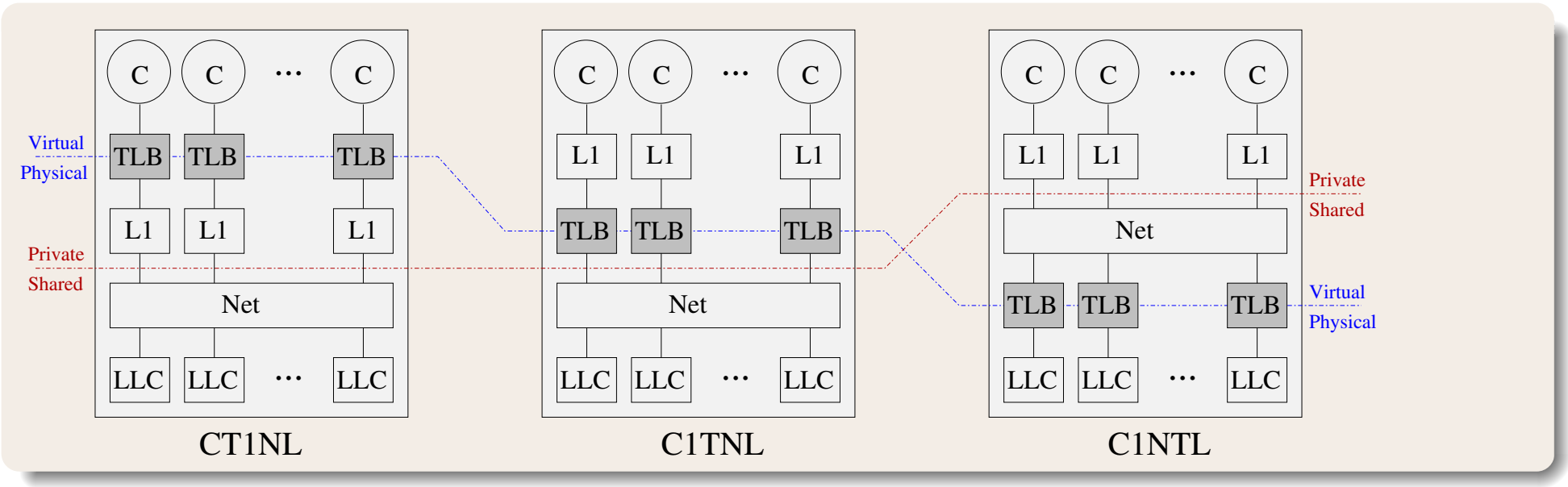
Solution for synonyms

VIPS-M solves the synonym problem:

- Synonyms for communication
  - Pages containing synonyms are classified as shared
  - Self-invalidation of shared data on synchronization points
- Synonyms for demaping/remaping
  - The OS self-invalidate all blocks belonging to a virtual page on eviction from main memory
- Synonyms in the same context
  - They are rare and explicitly requested by the user
  - Pages are set to read-only and writes trap to the OS where they are resolved

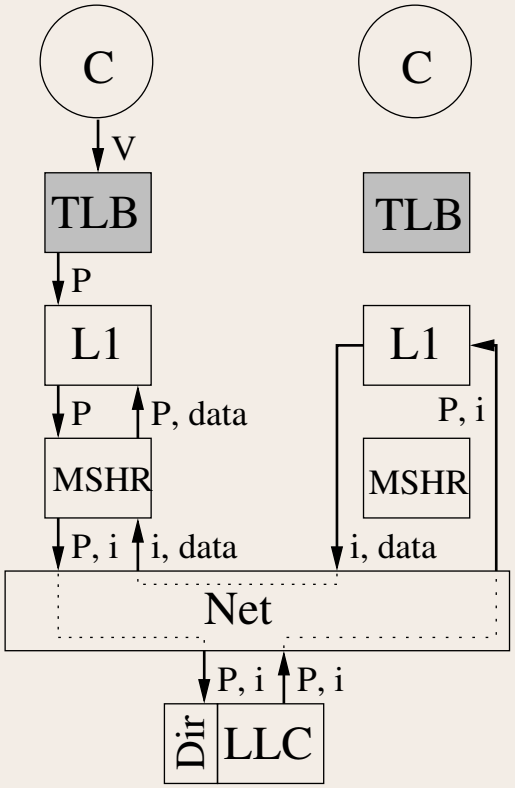
Design choices for TLB placement

- cT1NL: Physically-tagged L1 caches
- c1TNL: Virtual L1 caches, private TLBs
- c1NTL: Virtual L1 caches, shared TLBs



cT1NL: Physically-tagged L1 caches

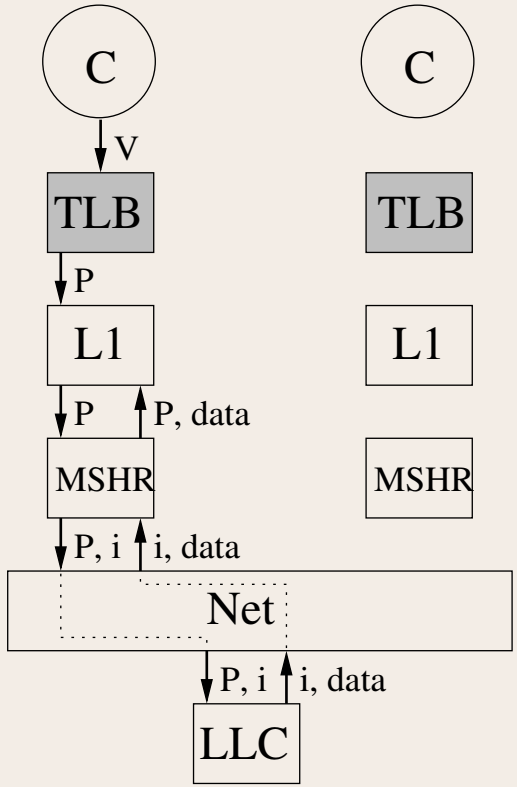
MESI



No reverse translation

High TLB power consumption

VIPS-M

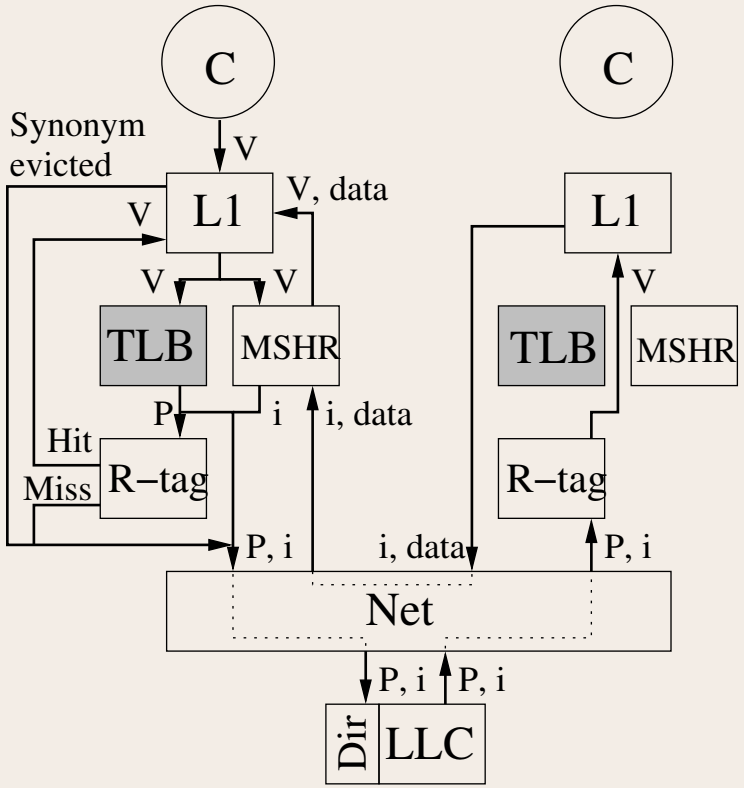


No reverse translation

High TLB power consumption

C1TNL: VIRTUAL L1 CACHES, PRIVATE TLBS

MESI

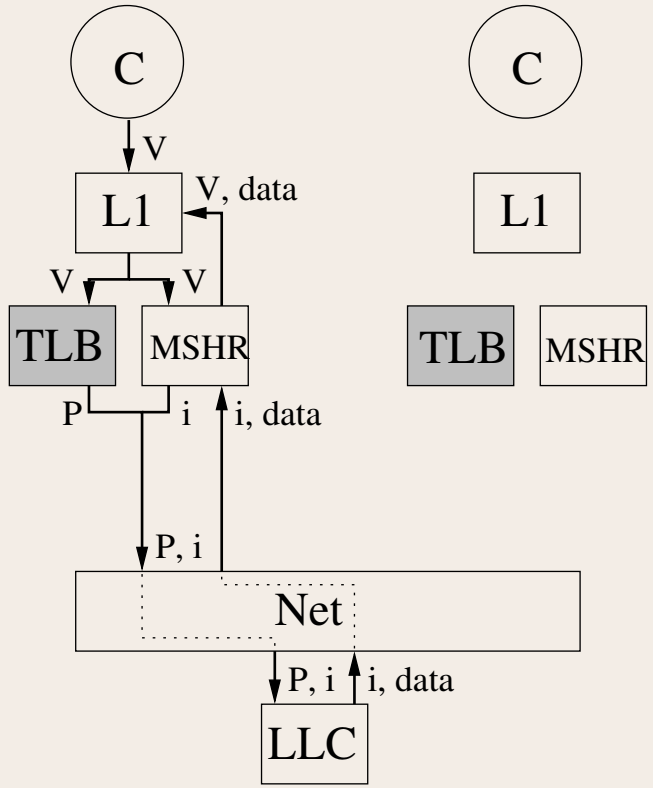


Low TLB power consumption

R-trans, syn-check

Replicated translations

VIPS-M



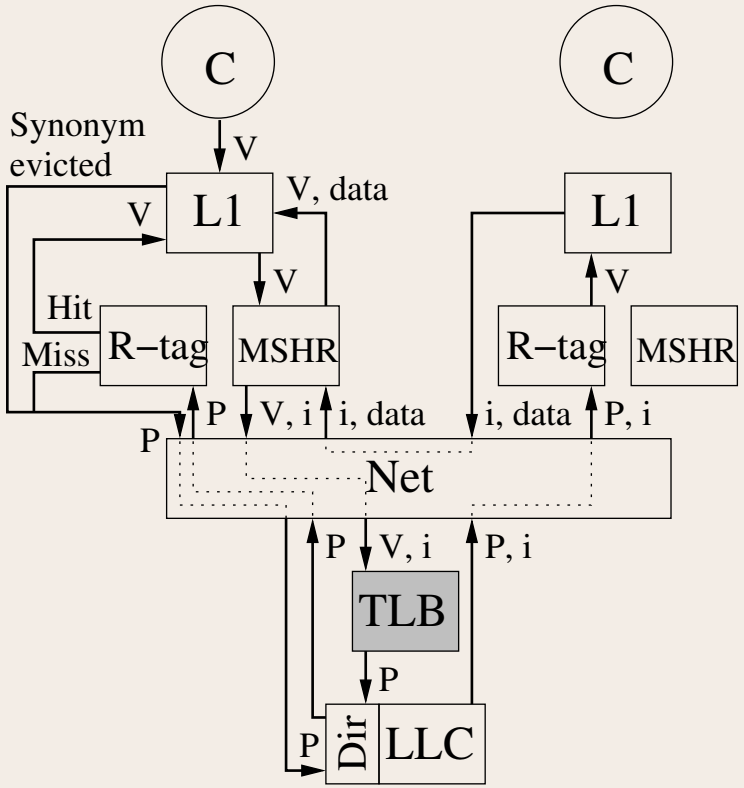
Low TLB power consumption

No r-trans, no syn-check

Replicated translations

C1NTL: VIRTUAL L1 CACHES, SHARED TLBS

MESI

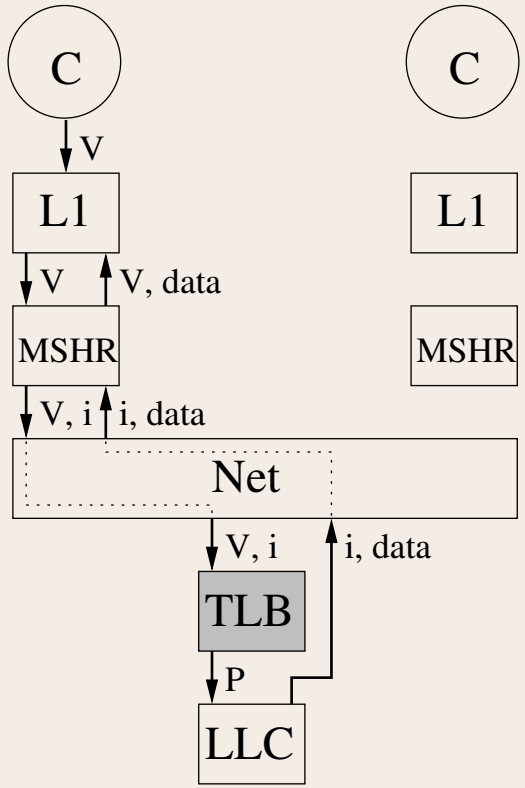


Low TLB power consumption

No replicated translations

R-trans, syn-check across Net

VIPS-M



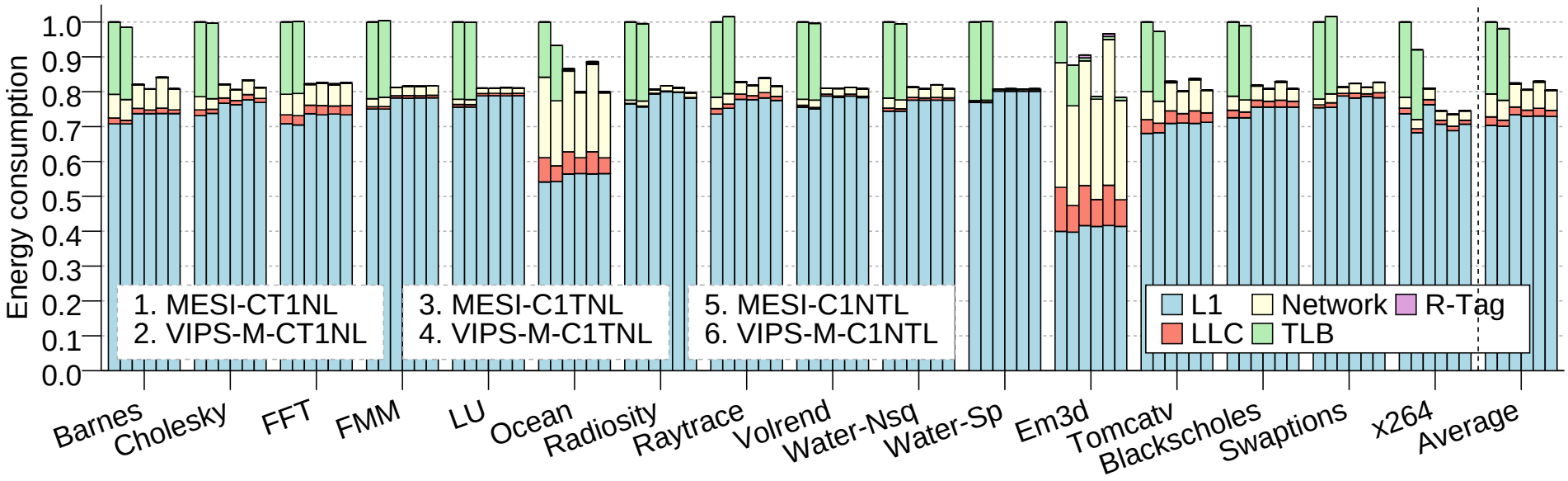
Low TLB power consumption

No replicated translations

No r-trans, no syn-check

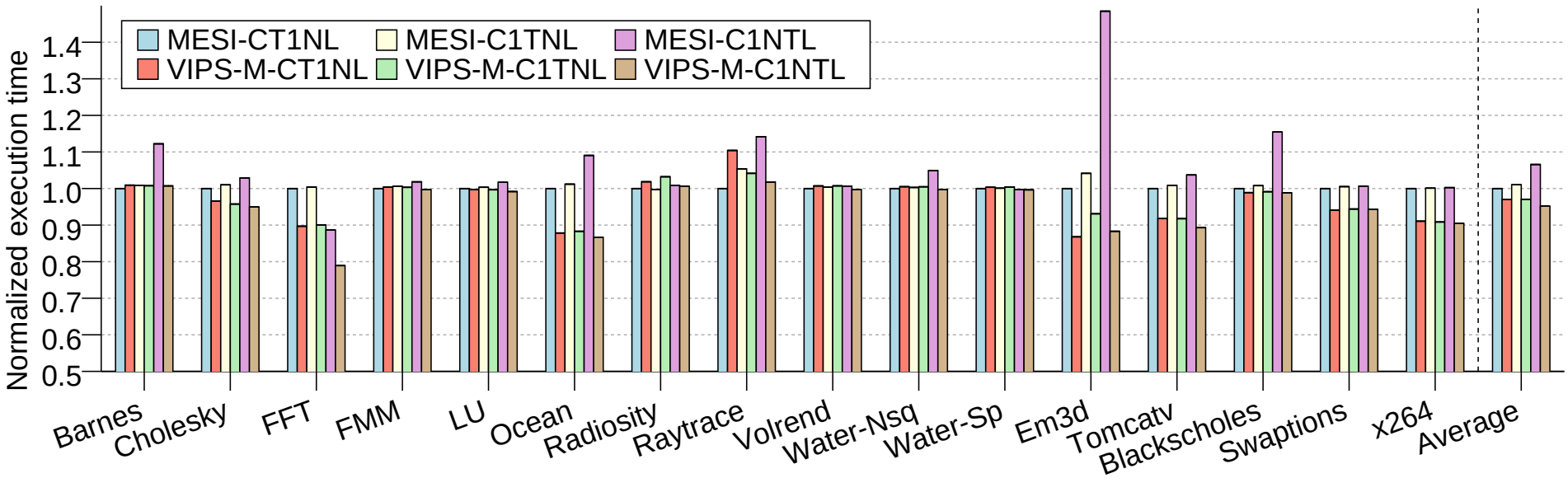
ENREGY CONSUMPTION

- Around 17% in energy reduction due to the use of virtual caches, mainly because of TLBs lookups
- VIPS-M keeps its advantage respect MESI (savings of 20% in total)
- The two VIPS-V protocols consume similar energy



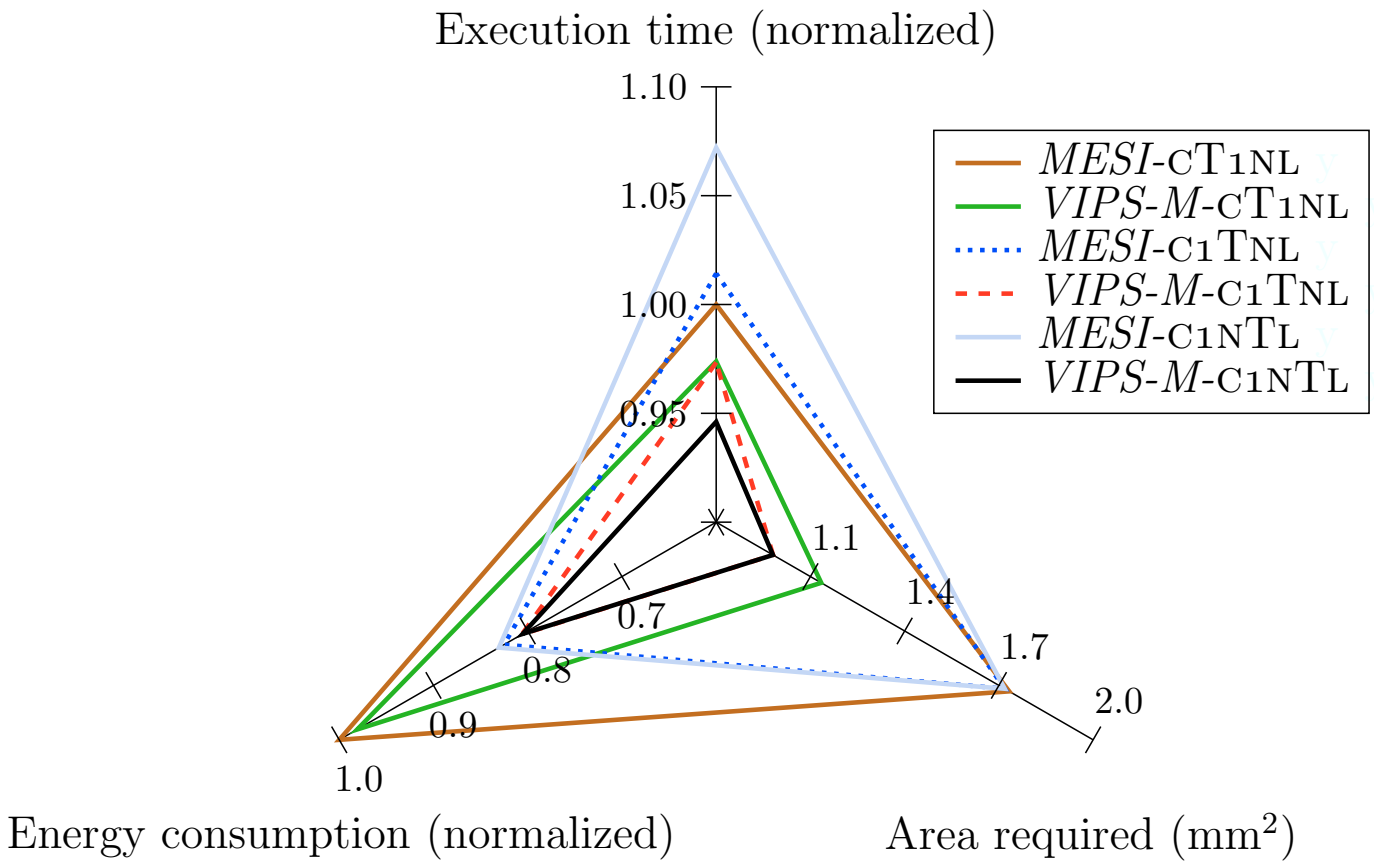
EXECUTION TIME

- Reverse translation is a problem for MESI protocols, especially for the shared TLB configuration
- For VIPS-M obtains improvements by sharing the TLB
- VIPS-V improves execution time by 5.4% w.r.t a physical MESI



TRADE-OFF GRAPH

- VIPS-M-C1NTL is the best in terms of performance, energy and area

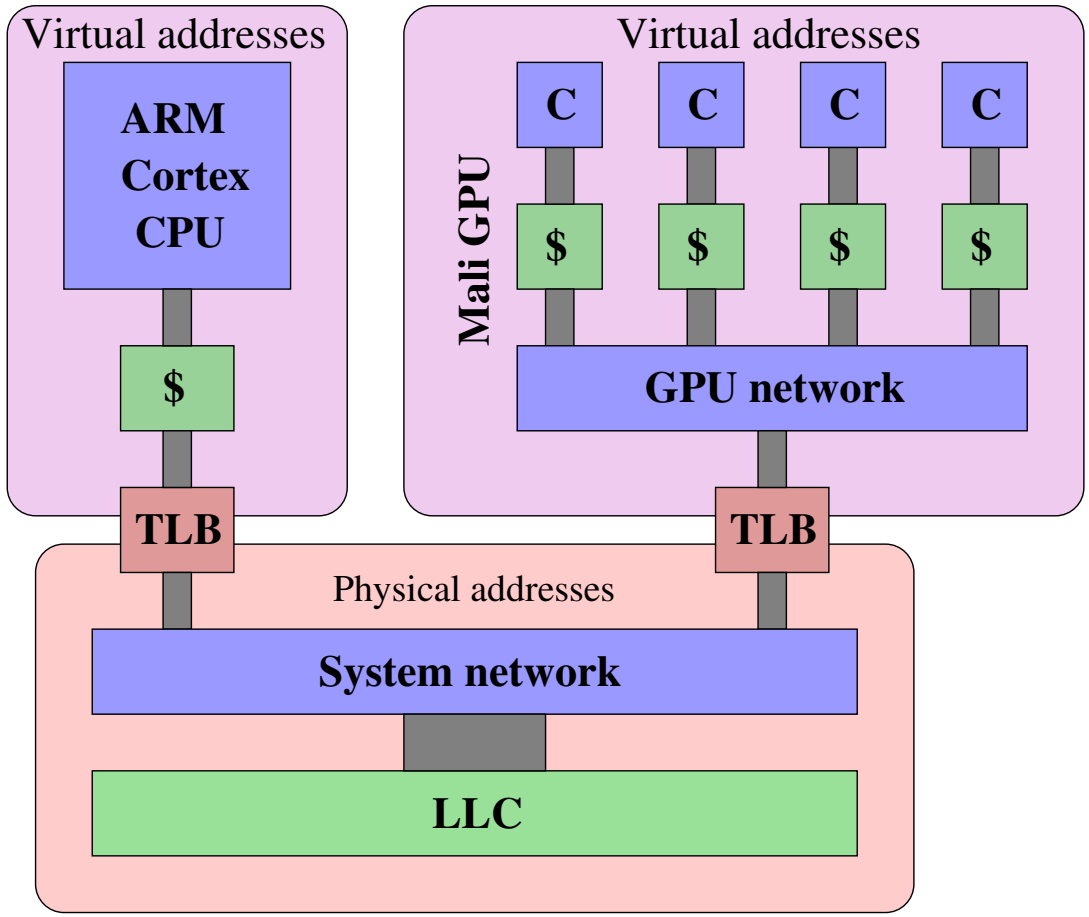


VIPS-V CONCLUSIONS

- We show that VIPS-M can implement virtual cache without reverse translations
- VIPS-M can simplify virtual cache coherence $\Rightarrow$ VIPS-V
- We obtain performance, energy, and area improvements with respect to a MESI protocol with physical caches
- VIPS-M allows different configurations for the placement of the TLB, which is ideal for CPU–GPU coherence

APPLICATION OF VIPS-V TO CPU&GPU COHERENCE

CPU: VIPS-M-C1TNL GPU: VIPS-M-C1NTL



Motivation

- Clustered cache hierarchies are a natural strategy for reducing the overhead introduced by cache coherence protocols (e.g., storage and traffic)[1]

[1] *Martin, Hill, and Sorin. "Why on-chip cache coherence is here to stay", CACM, 2012.*

Motivation

- Clustered cache hierarchies are a natural strategy for reducing the overhead introduced by cache coherence protocols (e.g., storage and traffic)[1]
- But clustered cache hierarchies bring another problem
  - ⇒ Design complexity: Keep the SWMR invariant in a clustered cache hierarchy
    - A *root* node sends invalidations and waits for acks
    - A *leaf* node receives an invalidations and answers with acks
    - An intermediate node in a hierarchy performs both actions ⇒ cross-product of states!
      (E.g., MOESI in GEMS: L1 → 16; L2 → 59; memory → 13)

[1] *Martin, Hill, and Sorin. "Why on-chip cache coherence is here to stay", CACM, 2012.*

Approach

- Simplify the source of complexity: invalidation/downgrade
- Inspired by recent proposals for simplifying flat coherence, DeNovo [Choi et al., PACT'11] and VIPS-M [Ros & Kaxiras, PACT'12]
  - No write-invalidation $\Rightarrow$ self-invalidation (SI) on synchronization points (VIPS-M & DeNovo)
  - No read-downgrade $\Rightarrow$ self-downgrade (SD) on synchronization points (VIPS-M)
  - Provide sequential consistency for data-race-free (SC for DRF) applications

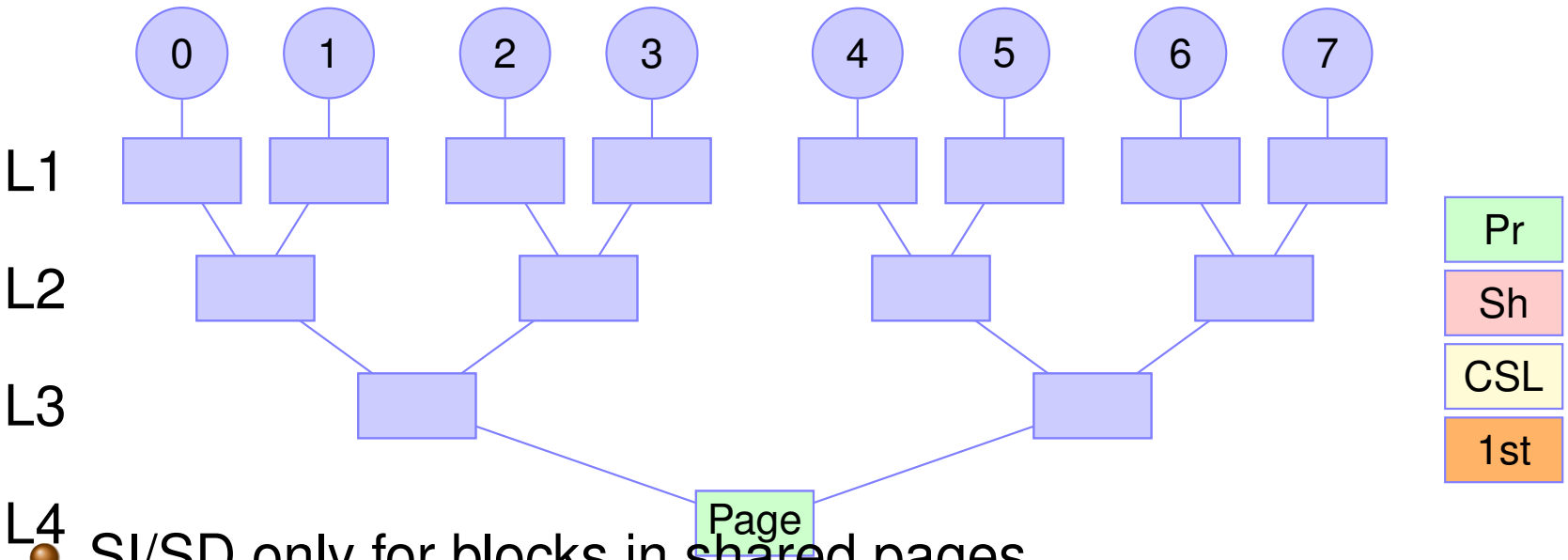
Challenge

- Our approach for simplifying the protocol is SI/SD
- A naïve implementation has to SI/SD all the data in the cache hierarchy
  - Not efficient!
- A new approach for restricting SI/SD in a clustered cache hierarchy is required

SOLUTION

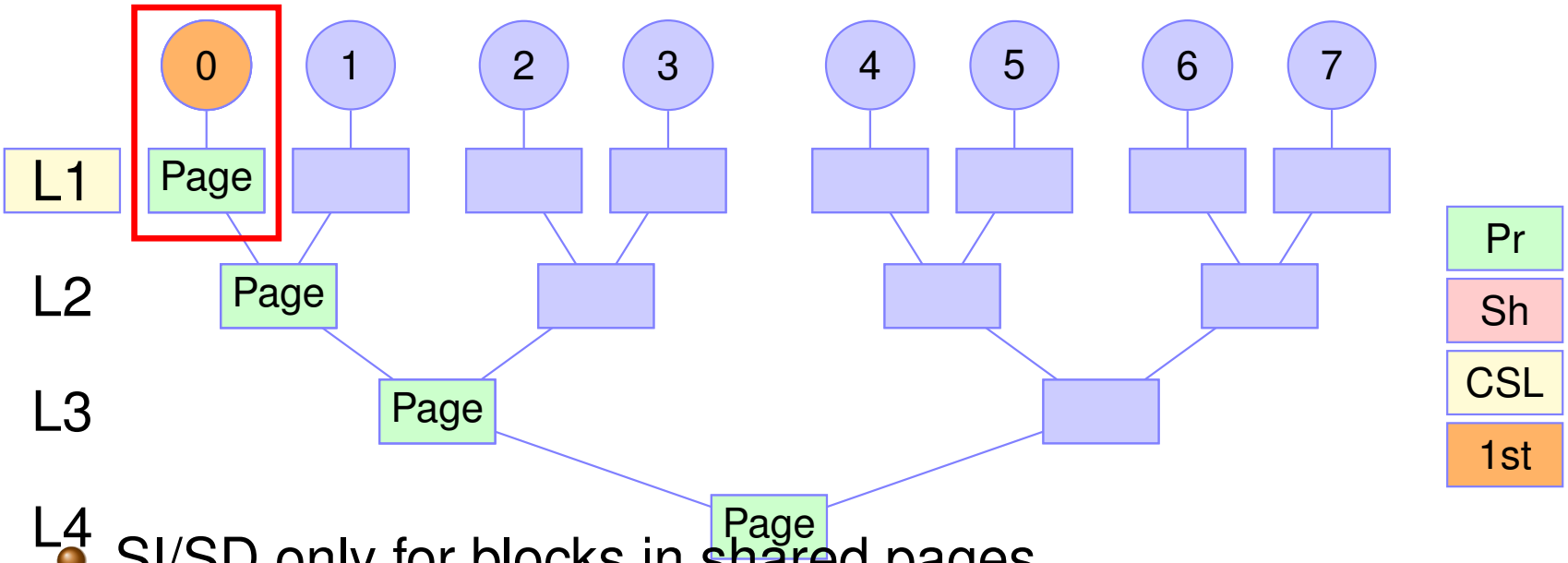
- We solve this problem by introducing the concept of hierarchical P/S classification
  - A block can be shared inside a cluster but be private outside
  - The level where this transition happens is the common sharing level (CSL)
  - Restrict SI/SD to shared blocks within a cluster
- Result
  - The protocol remains simple $\Rightarrow$ NO hierarchical complexity
  - Hierarchical complexity transferred to classification
  - In this paper we do classification at page level by adding information to the page tables
    - So all complexity is transferred to software

COMMON SHARING LEVEL



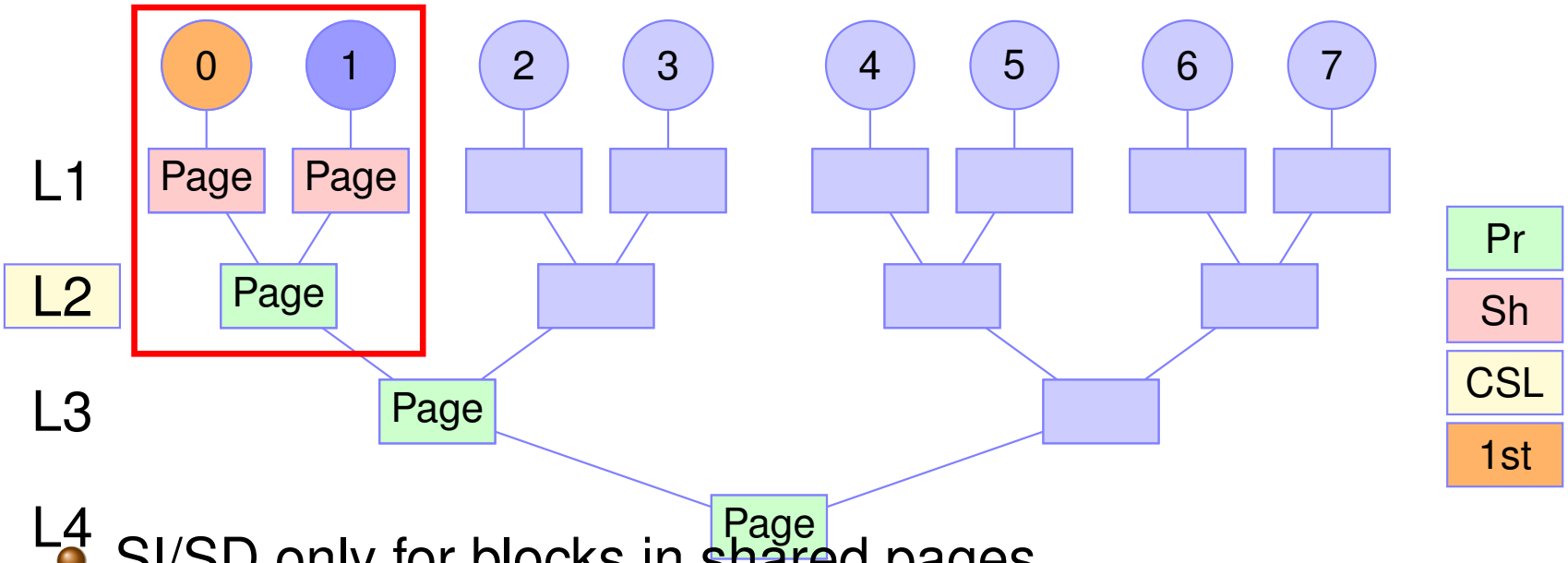
- SI/SD only for blocks in shared pages

COMMON SHARING LEVEL



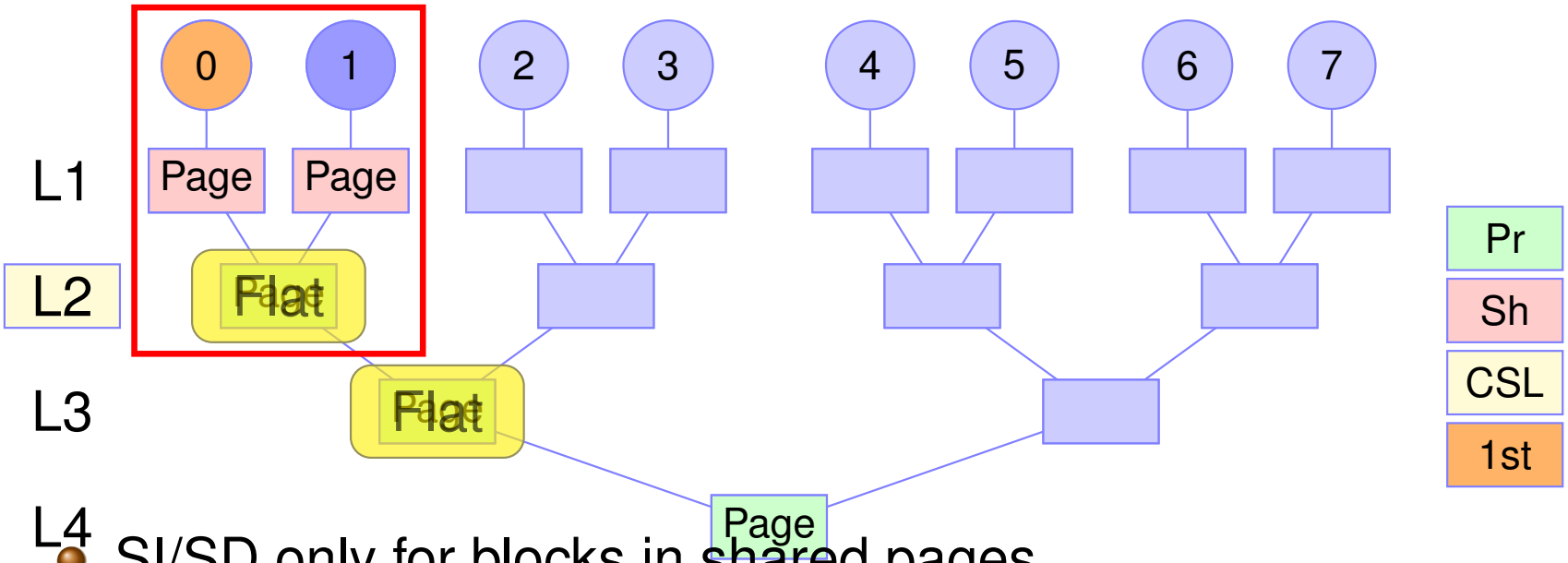
- SI/SD only for blocks in shared pages

COMMON SHARING LEVEL



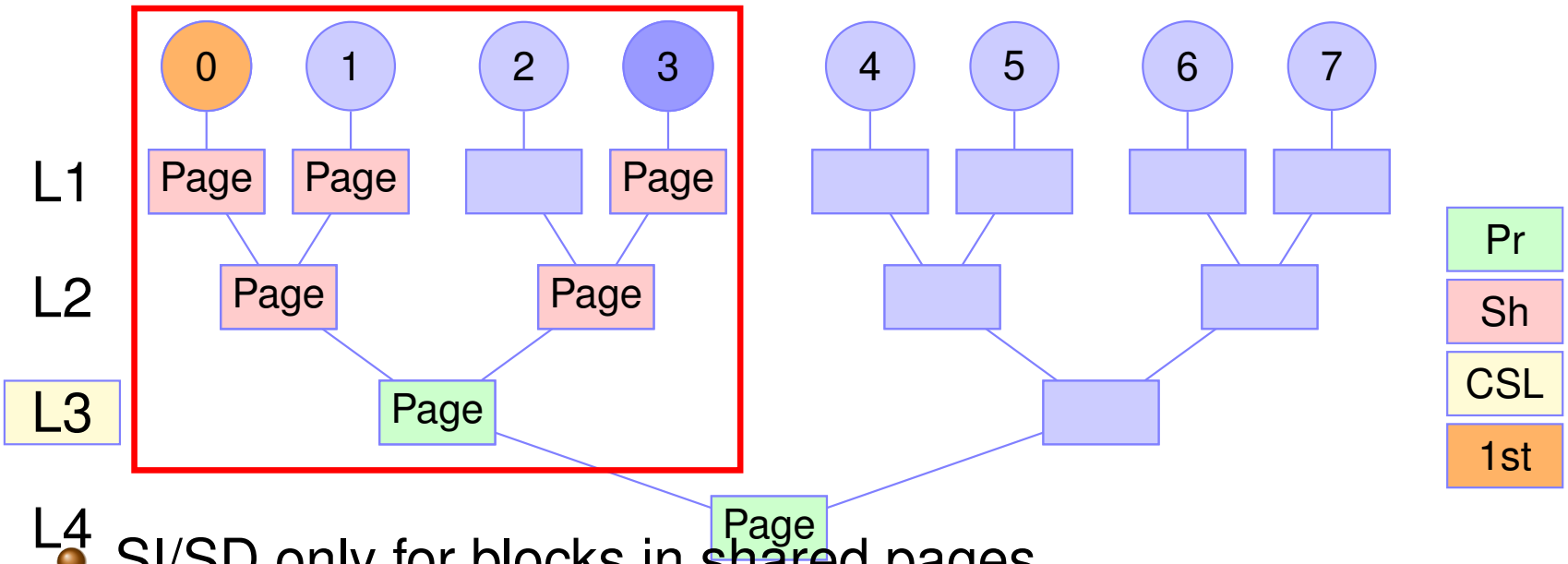
- SI/SD only for blocks in shared pages

COMMON SHARING LEVEL



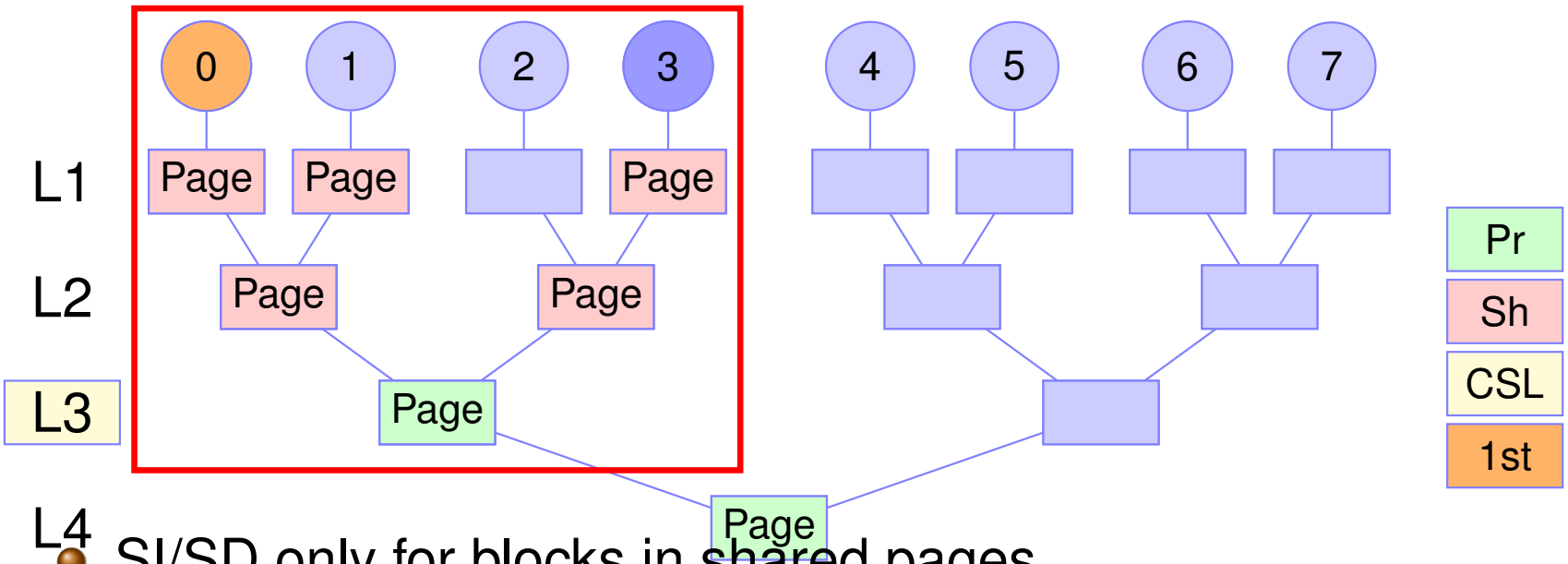
- SI/SD only for blocks in shared pages

COMMON SHARING LEVEL



- SI/SD only for blocks in shared pages

COMMON SHARING LEVEL



- SI/SD only for blocks in shared pages
- Page table entry (global hierarchy knowledge) stores:
  - First requester of a page ($log_2 N$)
  - CSL ($\lceil \log_2 \lceil \log_2 N / \log_2 d \rceil \rceil$): Root of the cluster containing all sharers
- TLB entry (local hierarchy knowledge) stores the CSL of the page
  - CSL is known before the cache miss takes place (a-priori)

VIPS-H: Events

- Keep frequent events simple and fast!
- Not frequent events more complex and slower

Events, frequency, complexity, and efficiency

Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
Load instr	21.3251%	16.4038%	High	Low	High
Store instr	6.8957%	5.4081%	High	Low	High
Atomic instr	0.0026%	0.0048%	Low	Low	Medium
SI&SD instr	0.0045%	0.0068%	Low	Medium	Low
Recoveries[1]	0.0006%	0.0005%	Very low	High	Low

[1] Performed upon changes in the CSL

EXAMPLE

EXAMPLE OF DRF CODE

```
/* Initially X & Y = 0 */
/* X & Y in the same block */
```

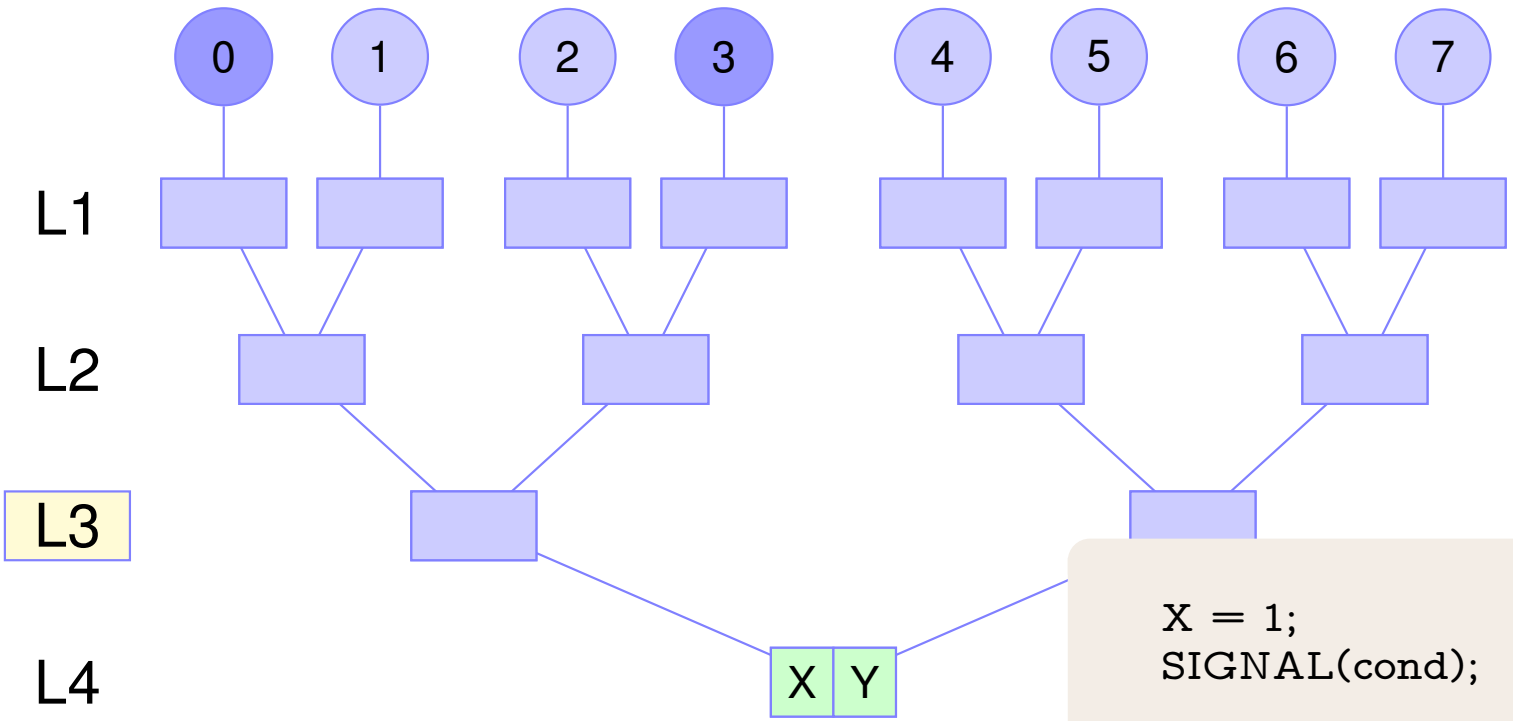
```
X = 1;
SIGNAL(cond);
```

```
$r2 = Y; /* False sharing; Prefetch X */
WAIT(cond);
$r1 = X; /* X should return 1 */
```

- Aquire and release semantics guarantee that read returns the value generated by the write
  - SIGNAL $\Rightarrow$ release semantics $\Rightarrow$ self-downgrade
  - WAIT $\Rightarrow$ aquire semantics $\Rightarrow$ self-invalidate

VIPS-H: LOAD OPERATIONS

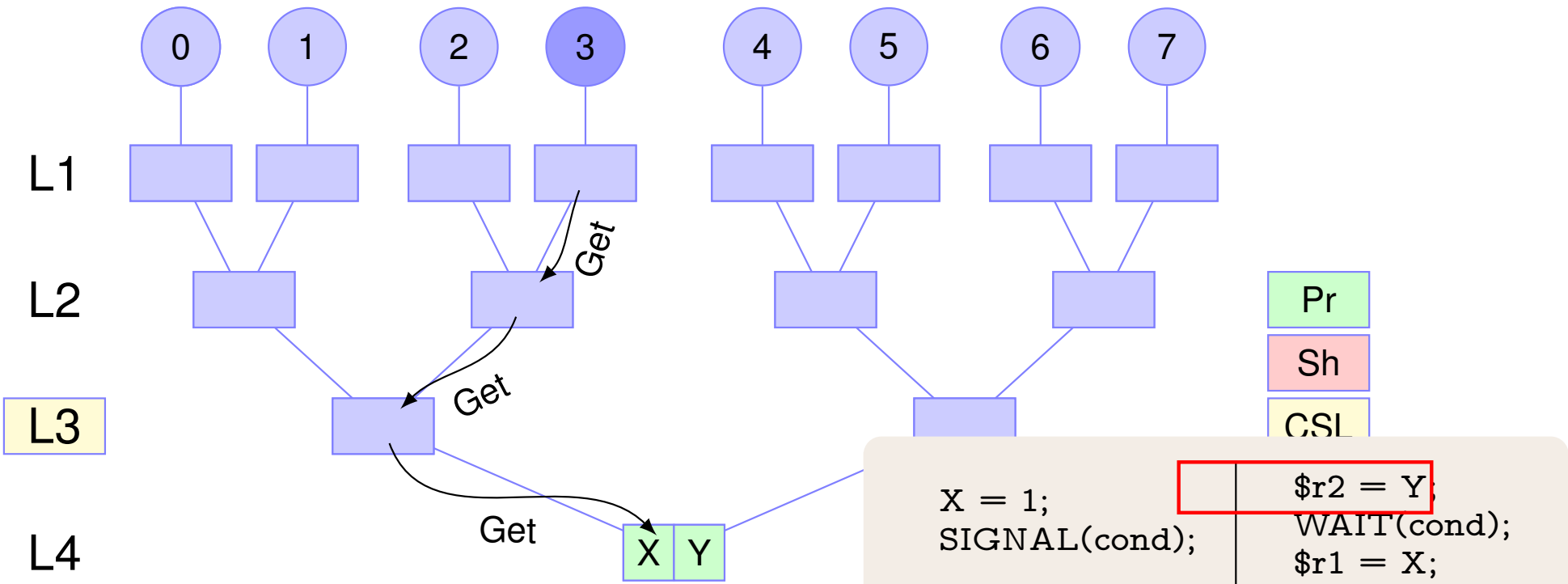
Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
Load instr	21.3251%	16.4038%	High	Low	High




```
X = 1;                  | $r2 = Y;
SIGNAL(cond);           | WAIT(cond);
                        | $r1 = X;
```

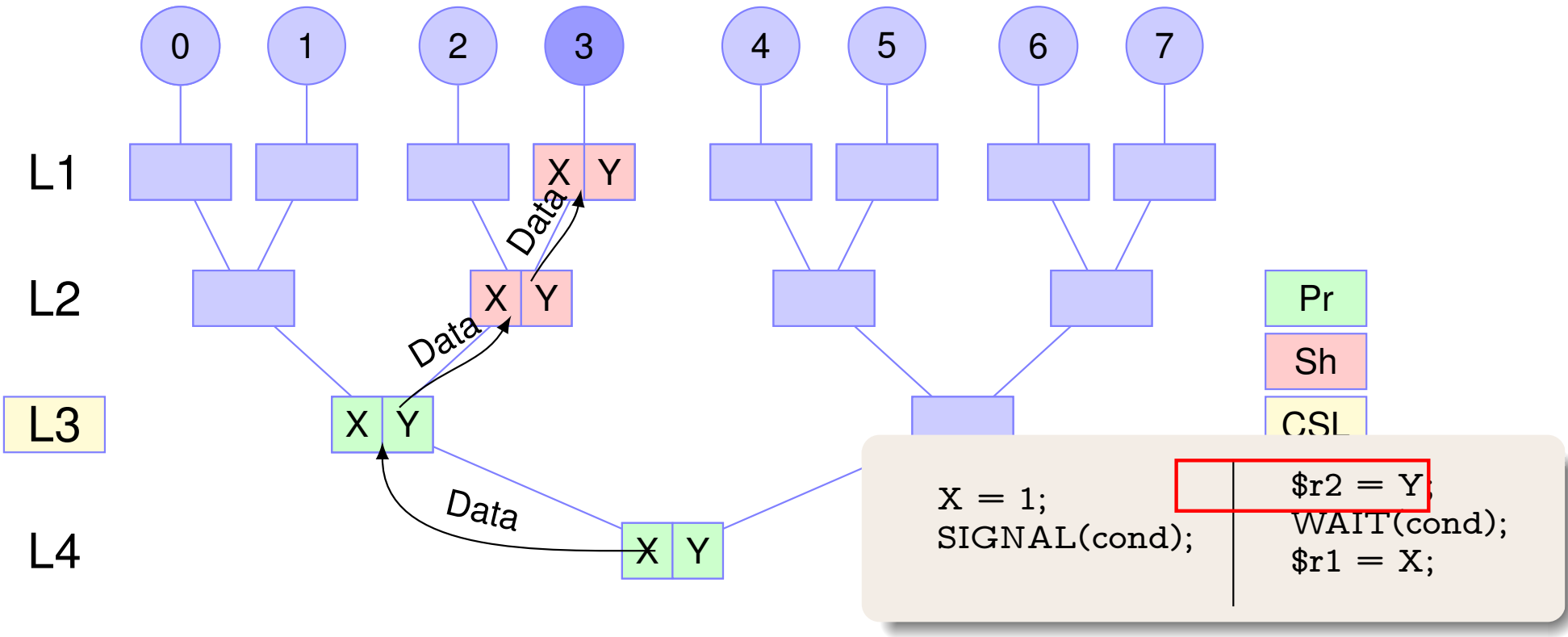
VIPS-H: LOAD OPERATIONS

Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
Load instr	21.3251%	16.4038%	High	Low	High



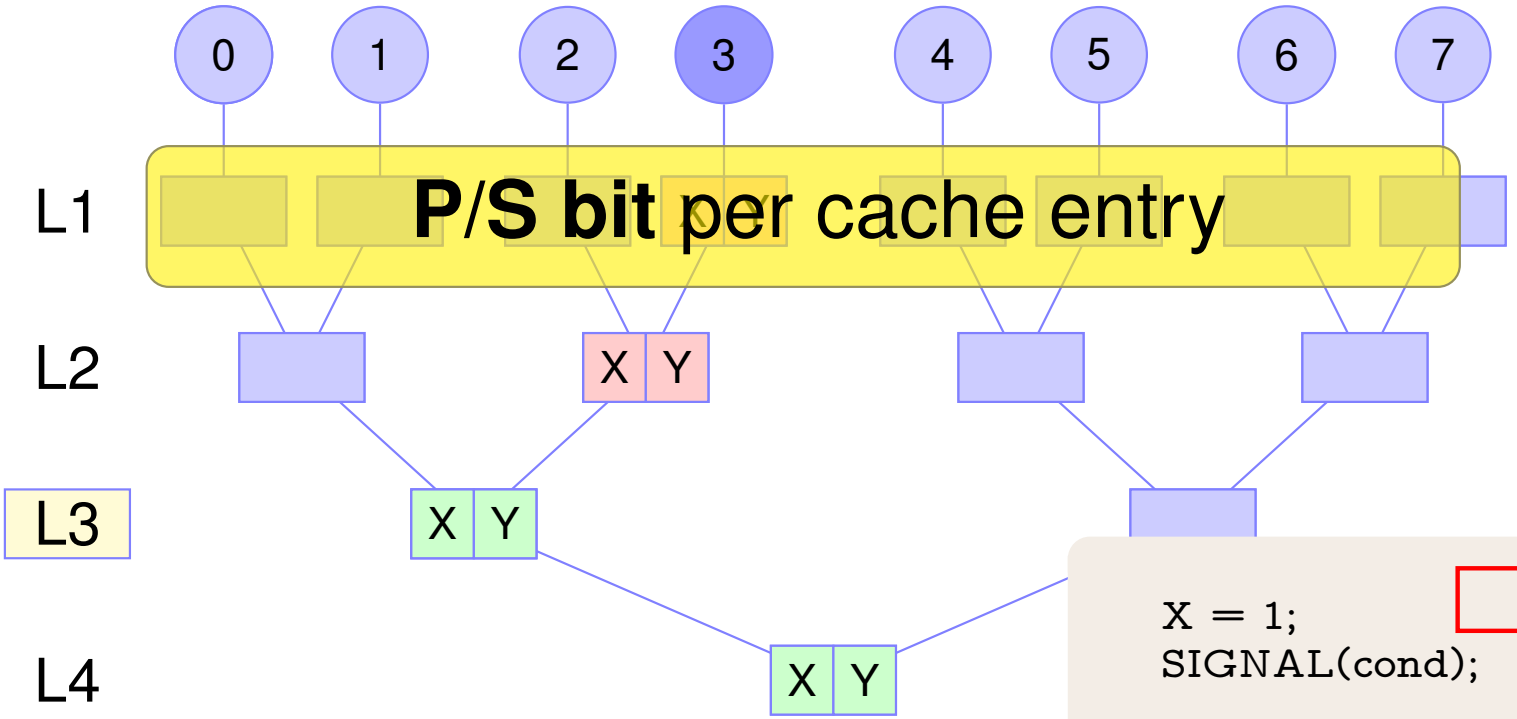
VIPS-H: LOAD OPERATIONS

Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
Load instr	21.3251%	16.4038%	High	Low	High



VIPS-H: LOAD OPERATIONS

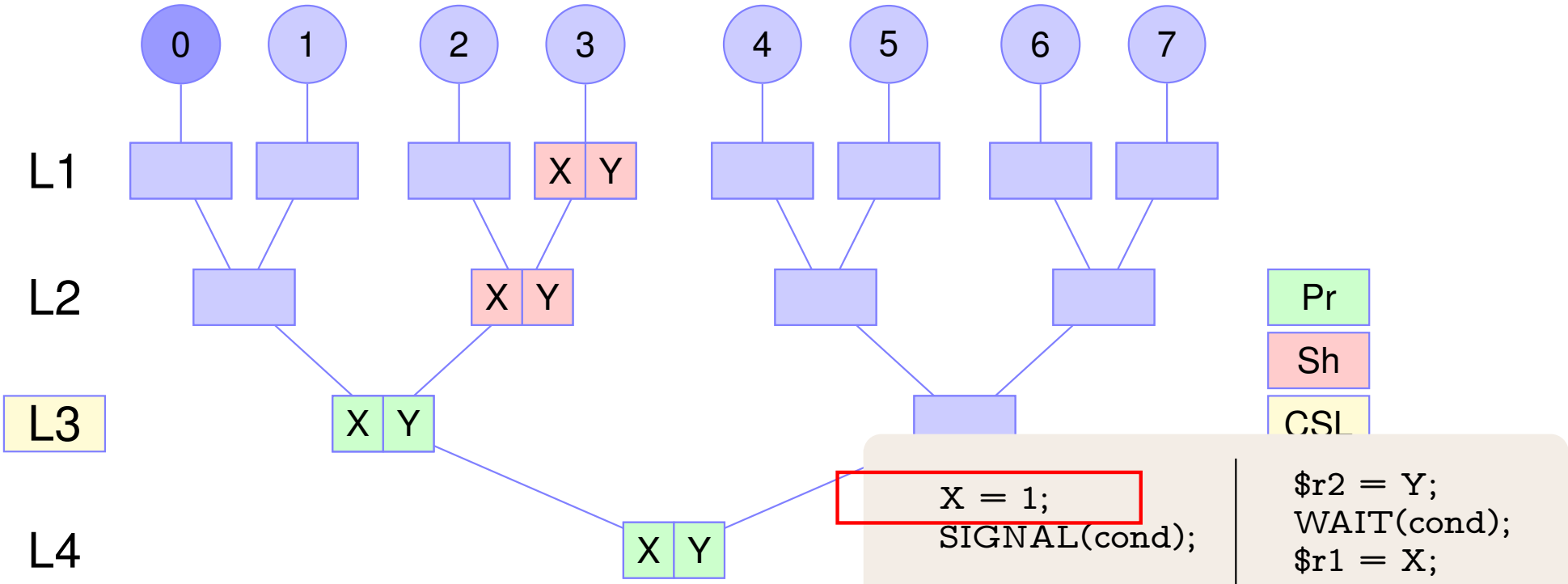
Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
Load instr	21.3251%	16.4038%	High	Low	High




```
X = 1;                  | $r2 = Y;
SIGNAL(cond);           | WAIT(cond);
                        | $r1 = X;
```

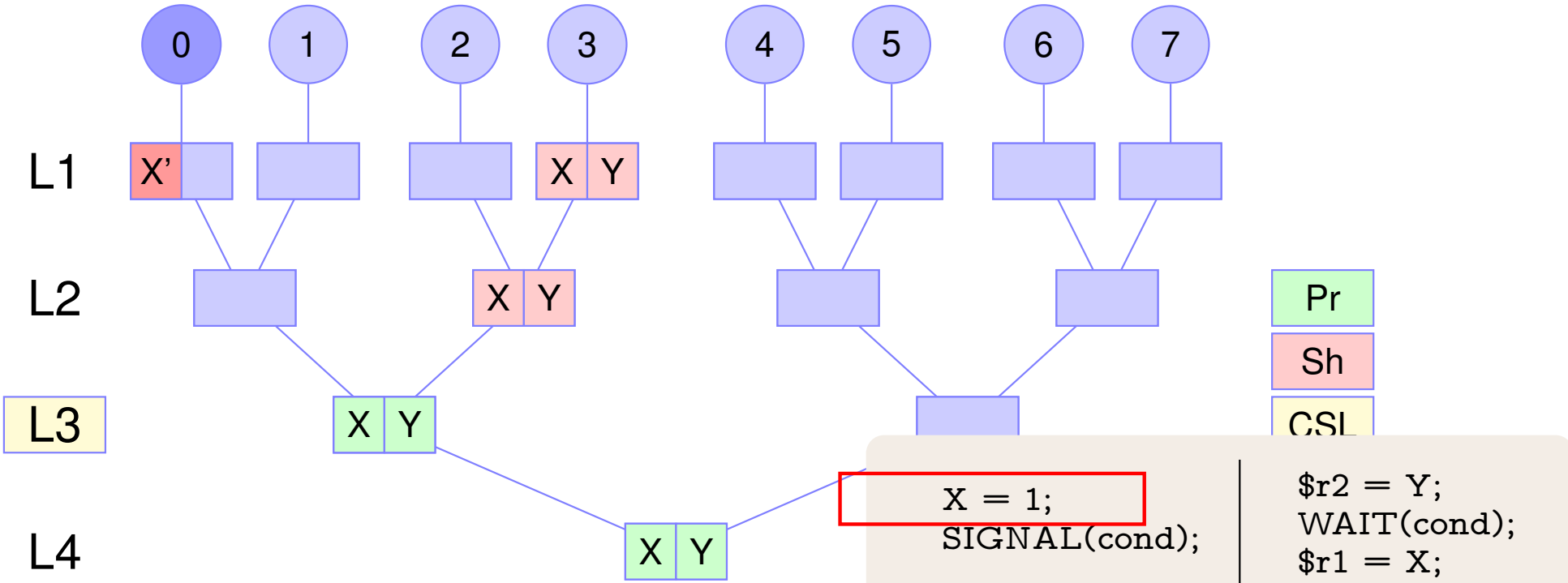
VIPS-H: STORE OPERATIONS

Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
Store instr	6.8957%	5.4081%	High	Low	High



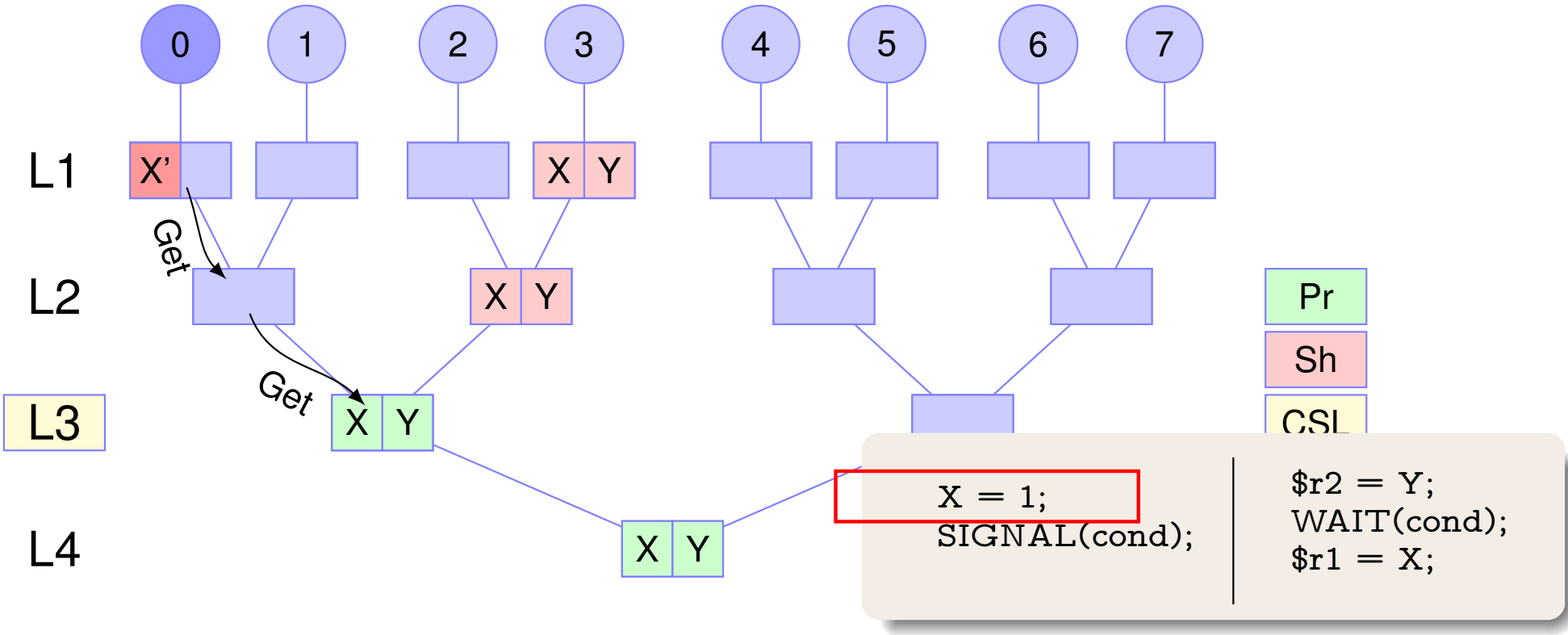
VIPS-H: STORE OPERATIONS

Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
Store instr	6.8957%	5.4081%	High	Low	High



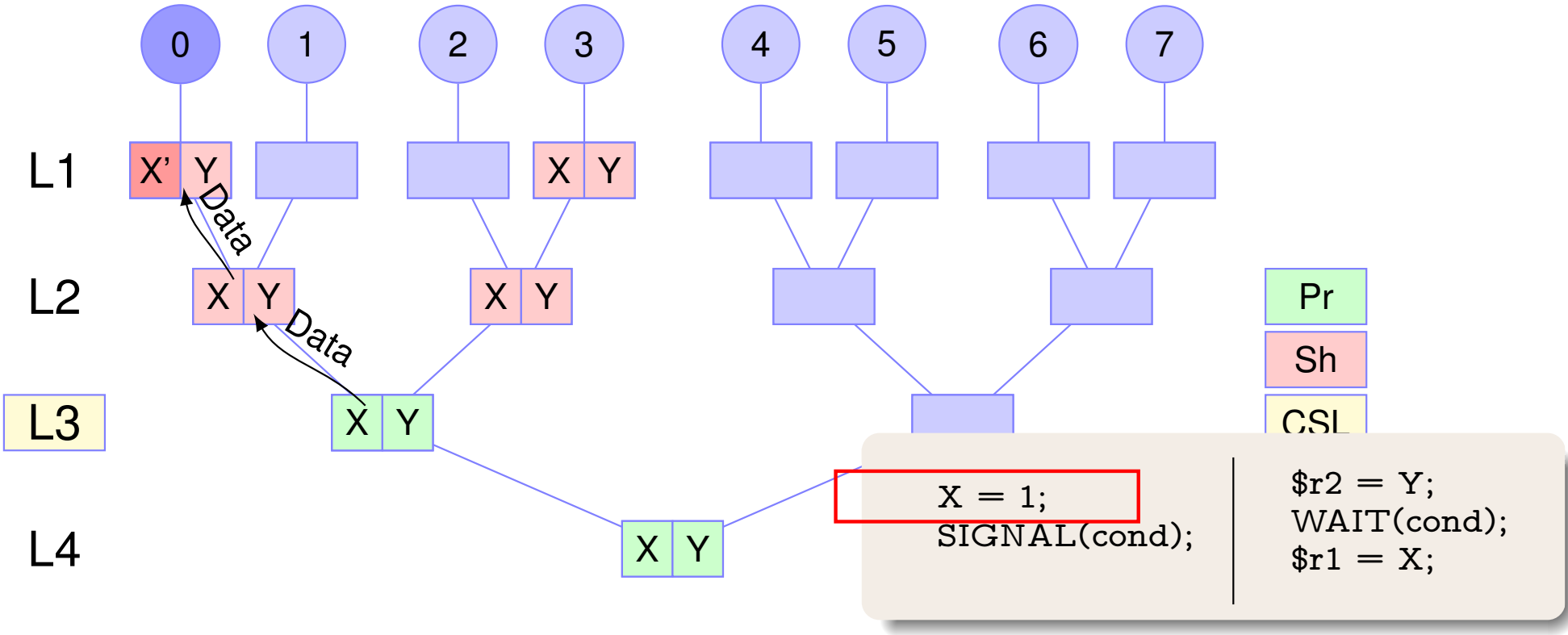
VIPS-H: STORE OPERATIONS

Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
Store instr	6.8957%	5.4081%	High	Low	High



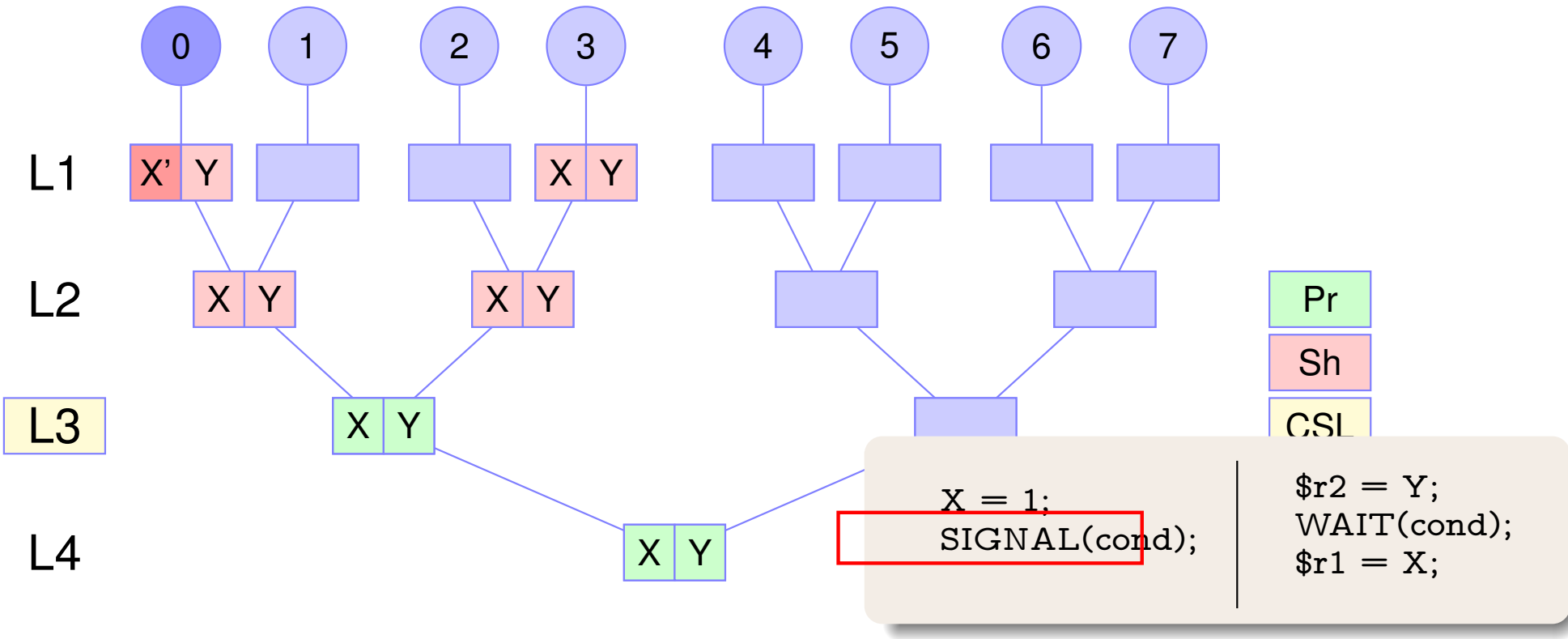
VIPS-H: STORE OPERATIONS

Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
Store instr	6.8957%	5.4081%	High	Low	High



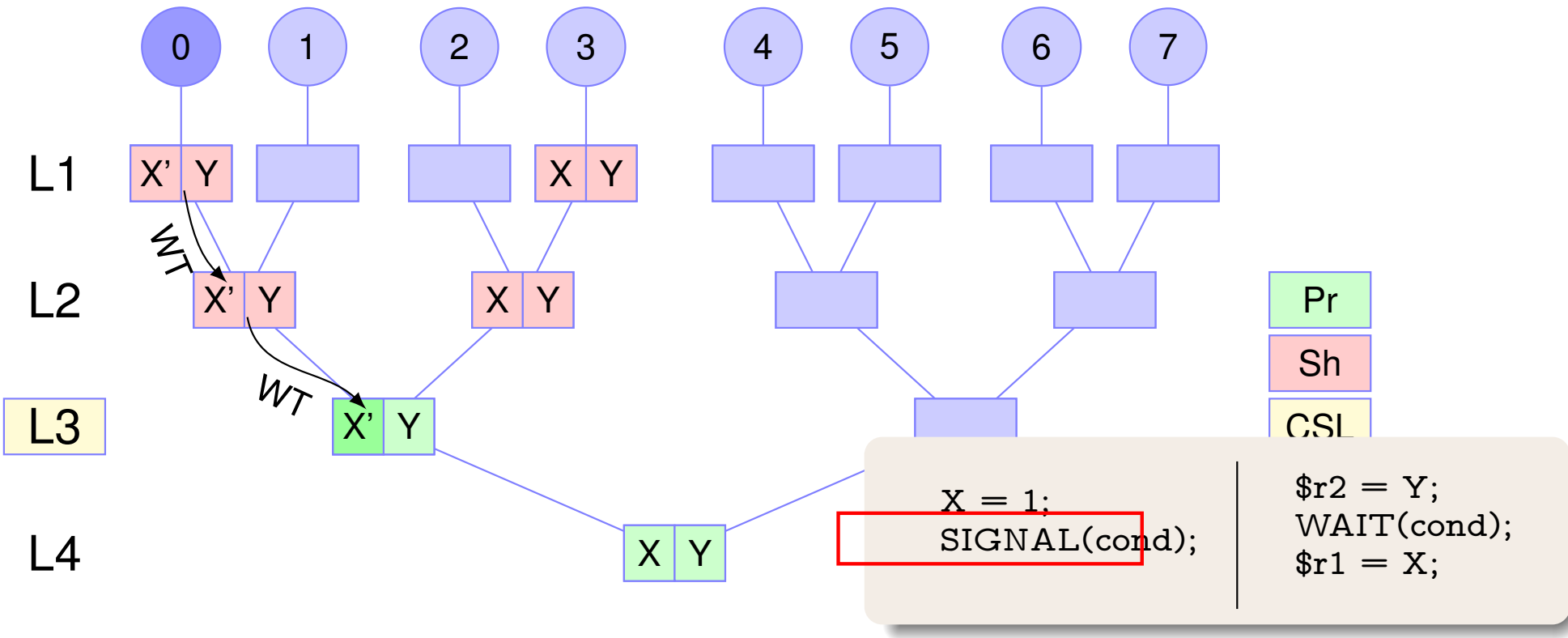
VIPS-H: SELF-DOWNGRADE

Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
SI&SD instr	0.0045%	0.0068%	Low	Medium	Low



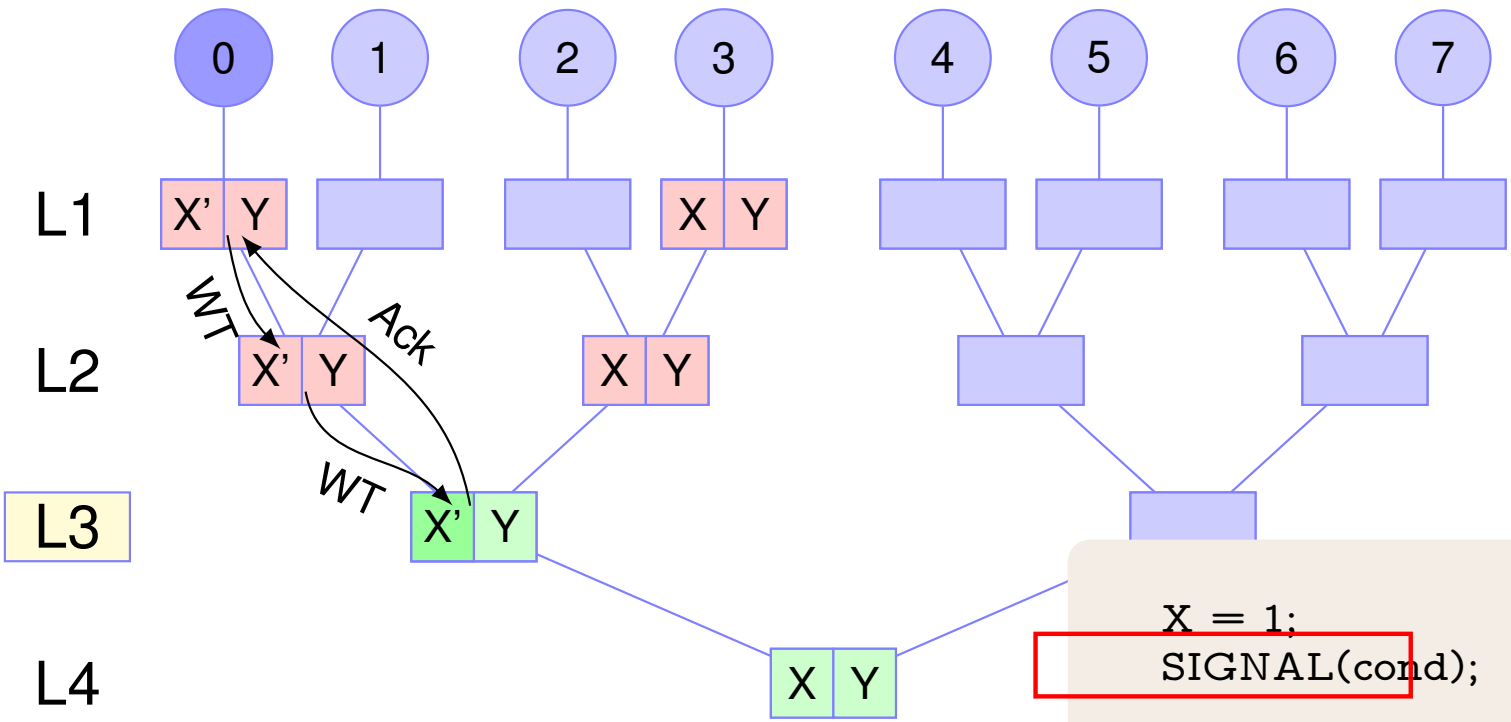
VIPS-H: SELF-DOWNGRADE

Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
SI&SD instr	0.0045%	0.0068%	Low	Medium	Low



VIPS-H: SELF-DOWNGRADE

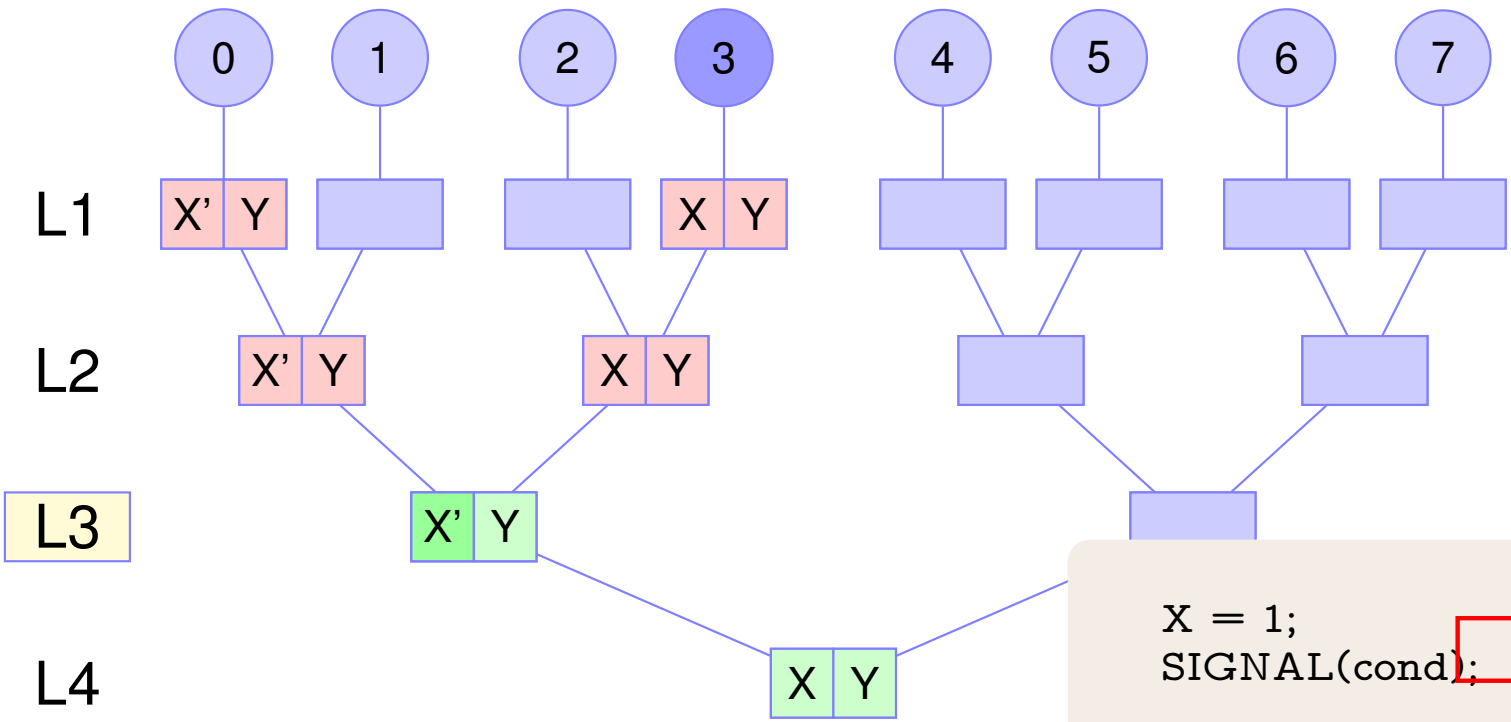
Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
SI&SD instr	0.0045%	0.0068%	Low	Medium	Low




```
X = 1;                  | $r2 = Y;
SIGNAL(cond);           | WAIT(cond);
                        | $r1 = X;
```

VIPS-H: SELF-INVALIDATION

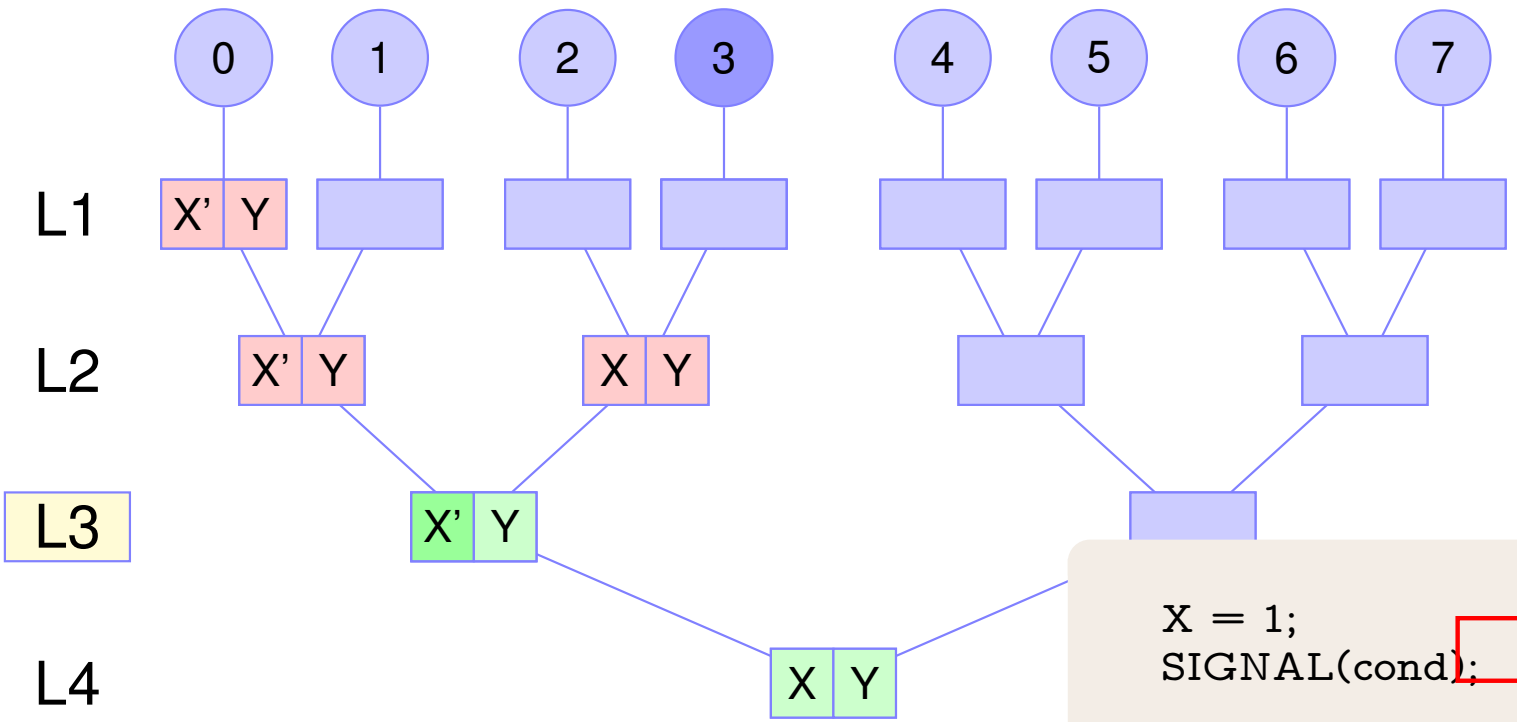
Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
SI&SD instr	0.0045%	0.0068%	Low	Medium	Low




```
X = 1;                  | $r2 = Y;
SIGNAL(cond);           | WAIT(cond);
                        | $r1 = X;
```

VIPS-H: SELF-INVALIDATION

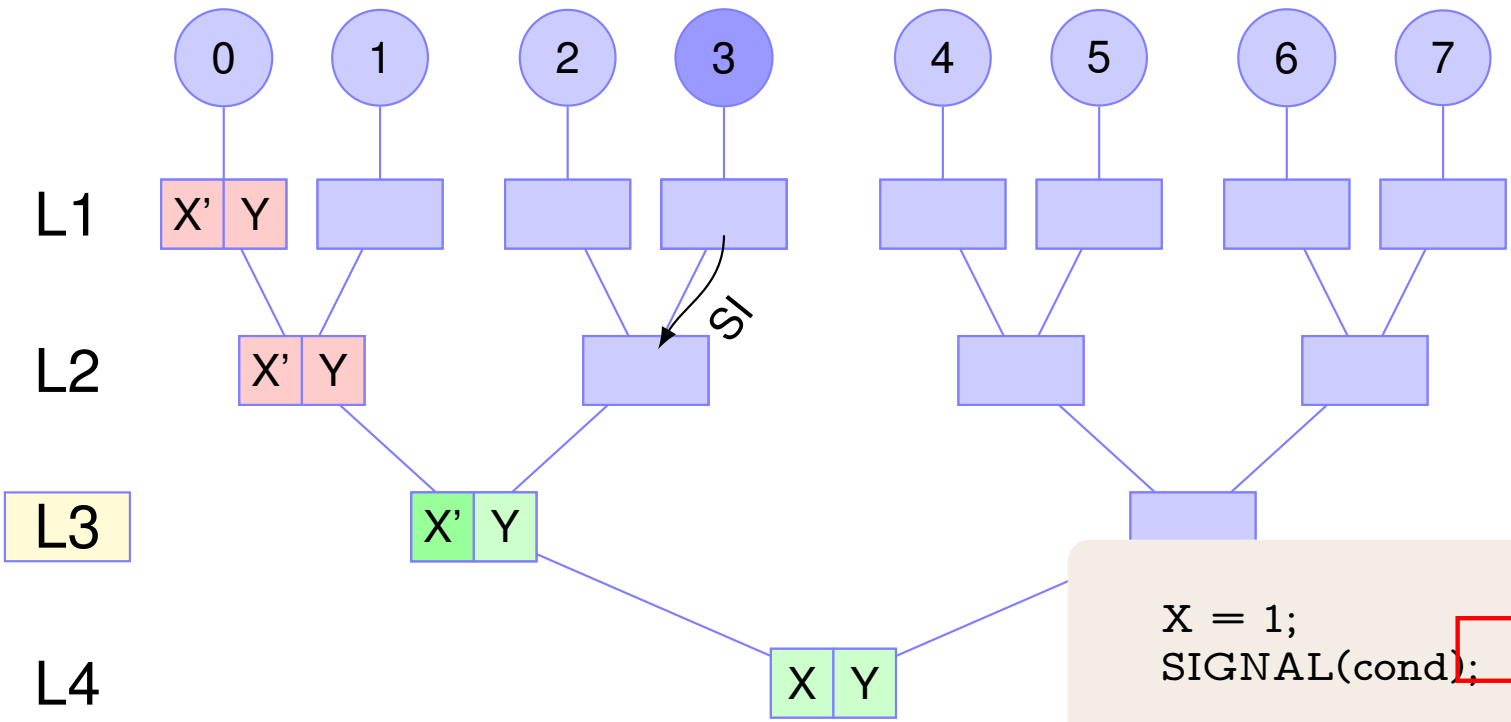
Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
SI&SD instr	0.0045%	0.0068%	Low	Medium	Low




```
X = 1;                  | $r2 = Y;
SIGNAL(cond);           | WAIT(cond);
                        | $r1 = X;
```

VIPS-H: SELF-INVALIDATION

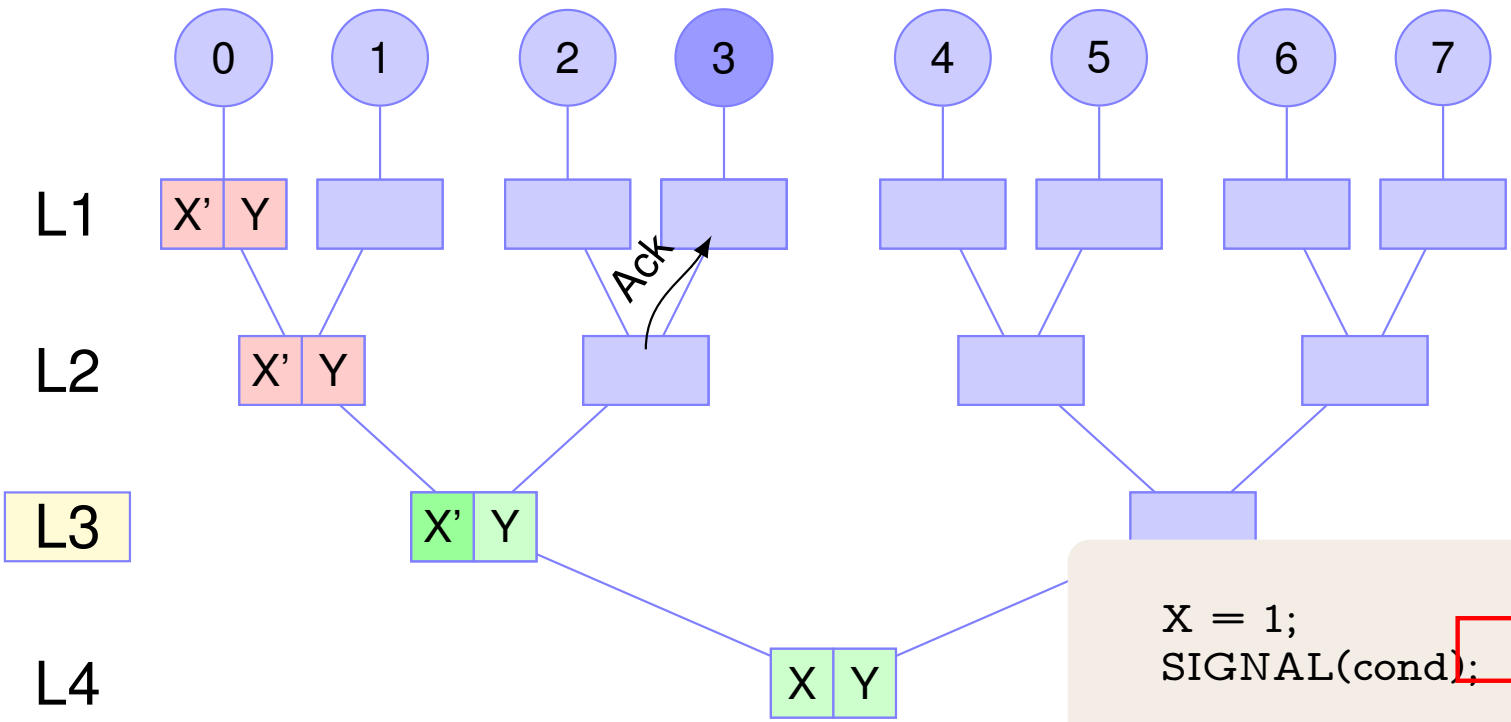
Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
SI&SD instr	0.0045%	0.0068%	Low	Medium	Low




```
X = 1;                  | $r2 = Y;
SIGNAL(cond);           | WAIT(cond);
                        | $r1 = X;
```

VIPS-H: SELF-INVALIDATION

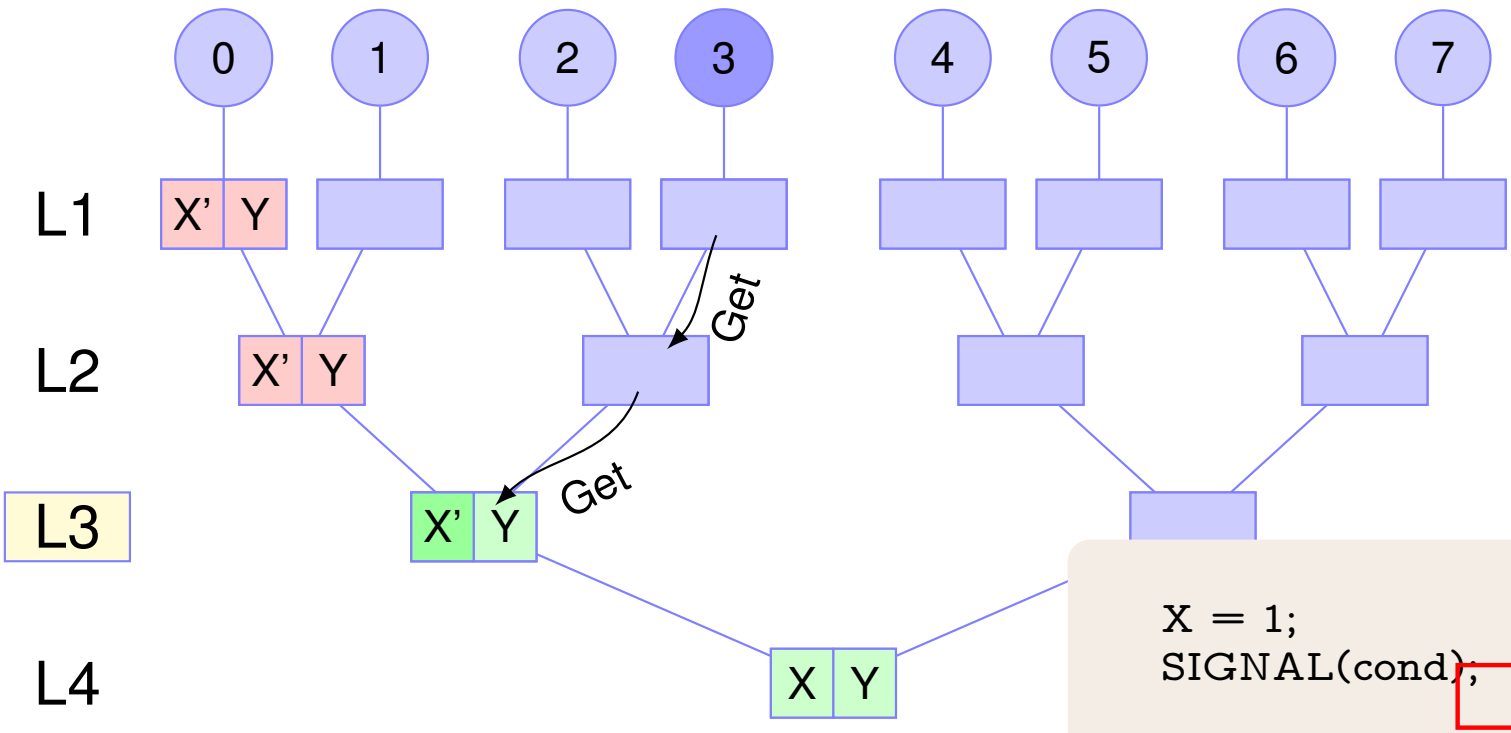
Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
SI&SD instr	0.0045%	0.0068%	Low	Medium	Low




```
X = 1;                  | $r2 = Y;
SIGNAL(cond);           | WAIT(cond);
                        | $r1 = X;
```

VIPS-H: SELF-INVALIDATION

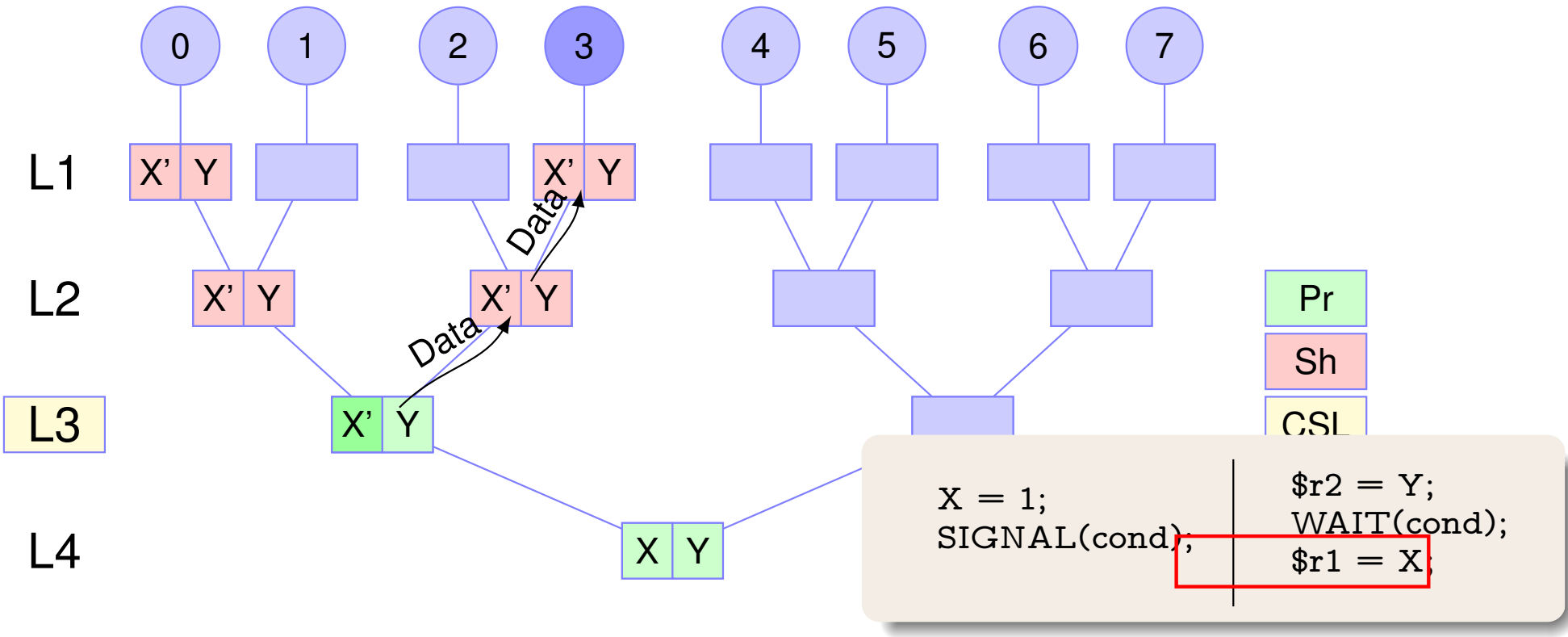
Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
SI&SD instr	0.0045%	0.0068%	Low	Medium	Low




```
X = 1;                    | $r2 = Y;
SIGNAL(cond);             | WAIT(cond);
                          | $r1 = X;
```

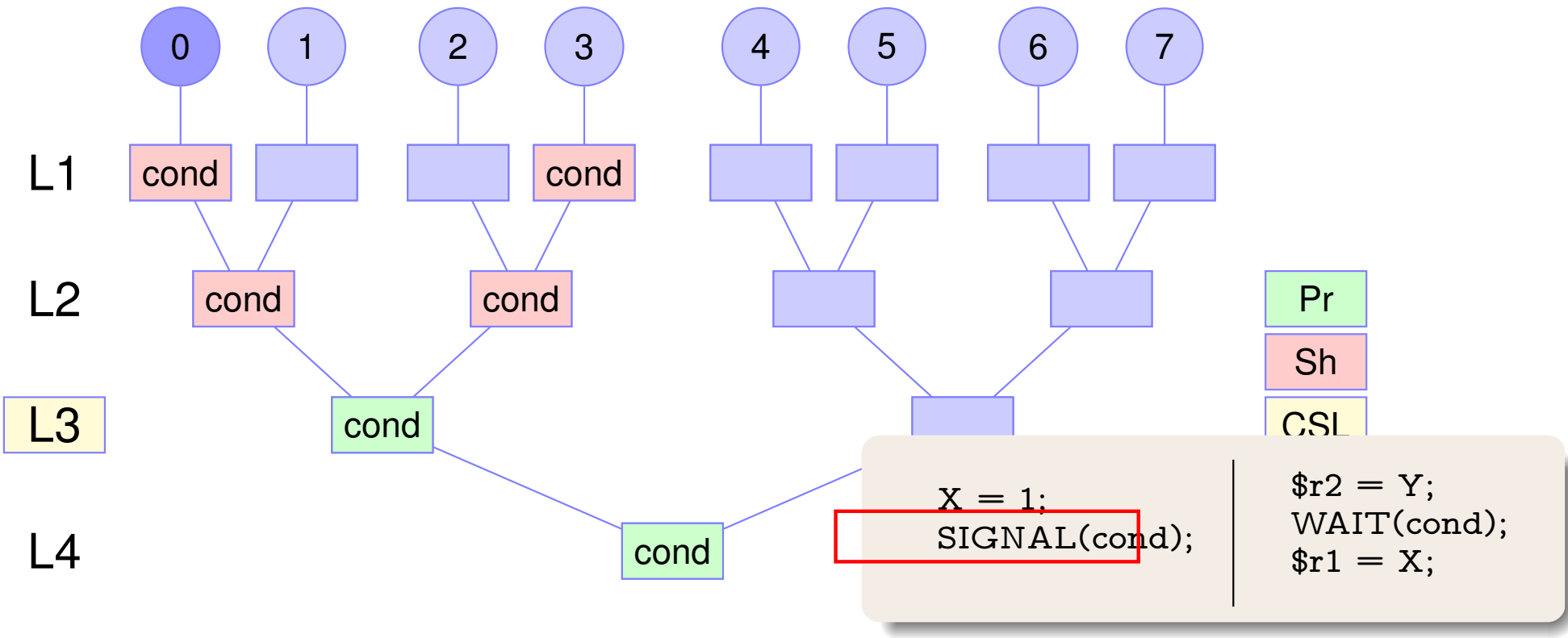
VIPS-H: SELF-INVALIDATION

Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
SI&SD instr	0.0045%	0.0068%	Low	Medium	Low



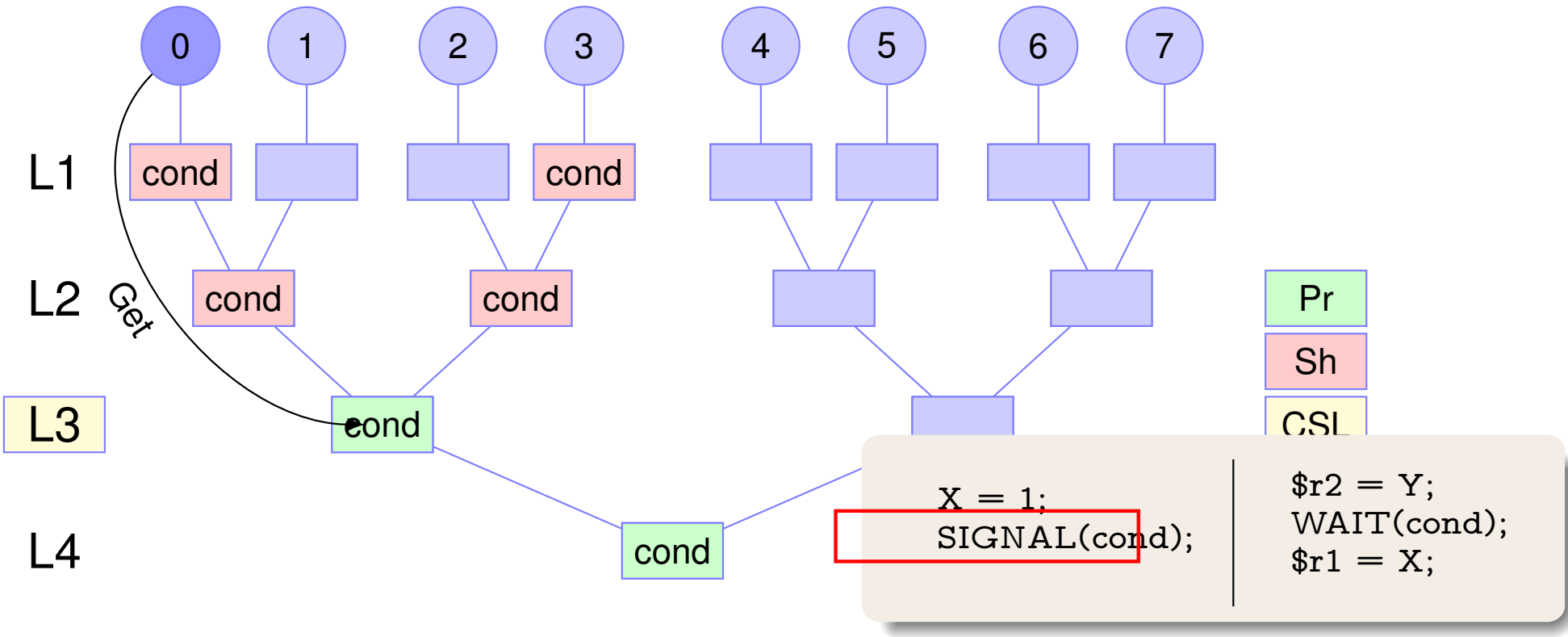
VIPS-H: ATOMIC OPERATIONS

Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
Atomic instr	0.0026%	0.0048%	Low	Low	Medium



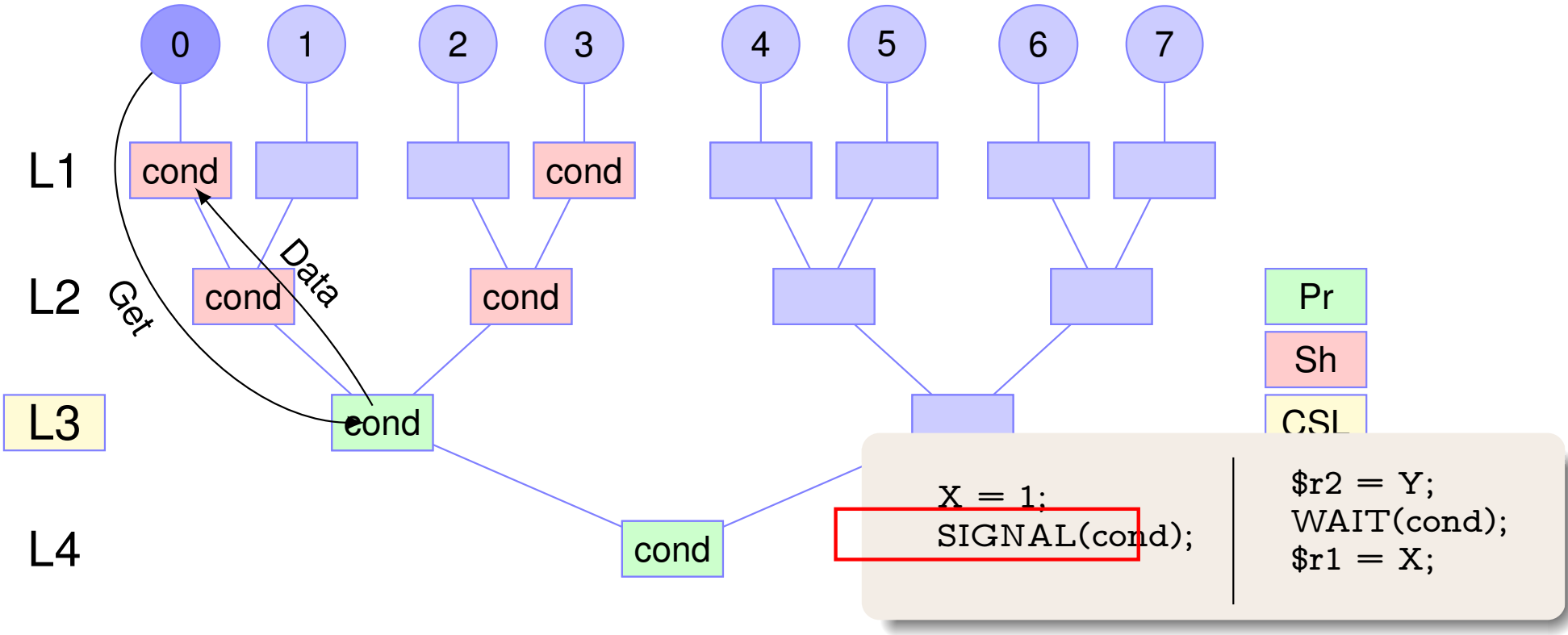
VIPS-H: ATOMIC OPERATIONS

Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
Atomic instr	0.0026%	0.0048%	Low	Low	Medium



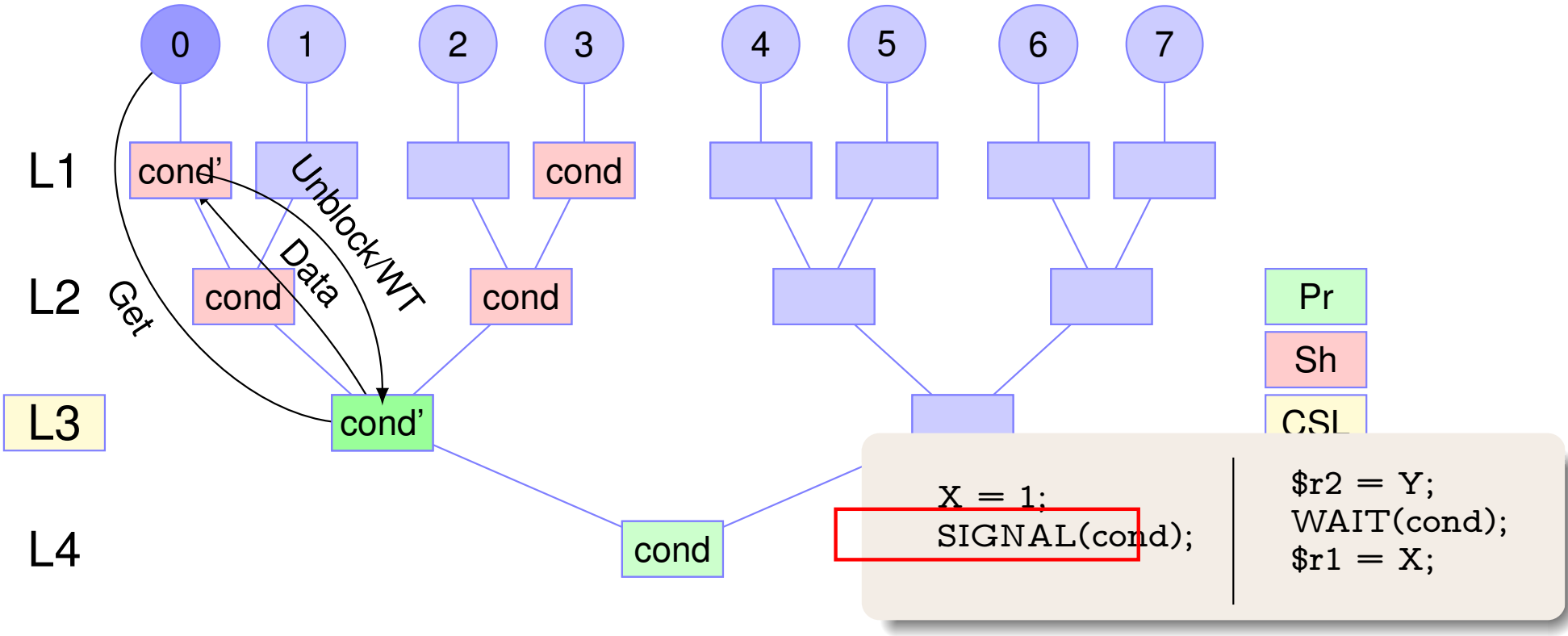
VIPS-H: ATOMIC OPERATIONS

Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
Atomic instr	0.0026%	0.0048%	Low	Low	Medium



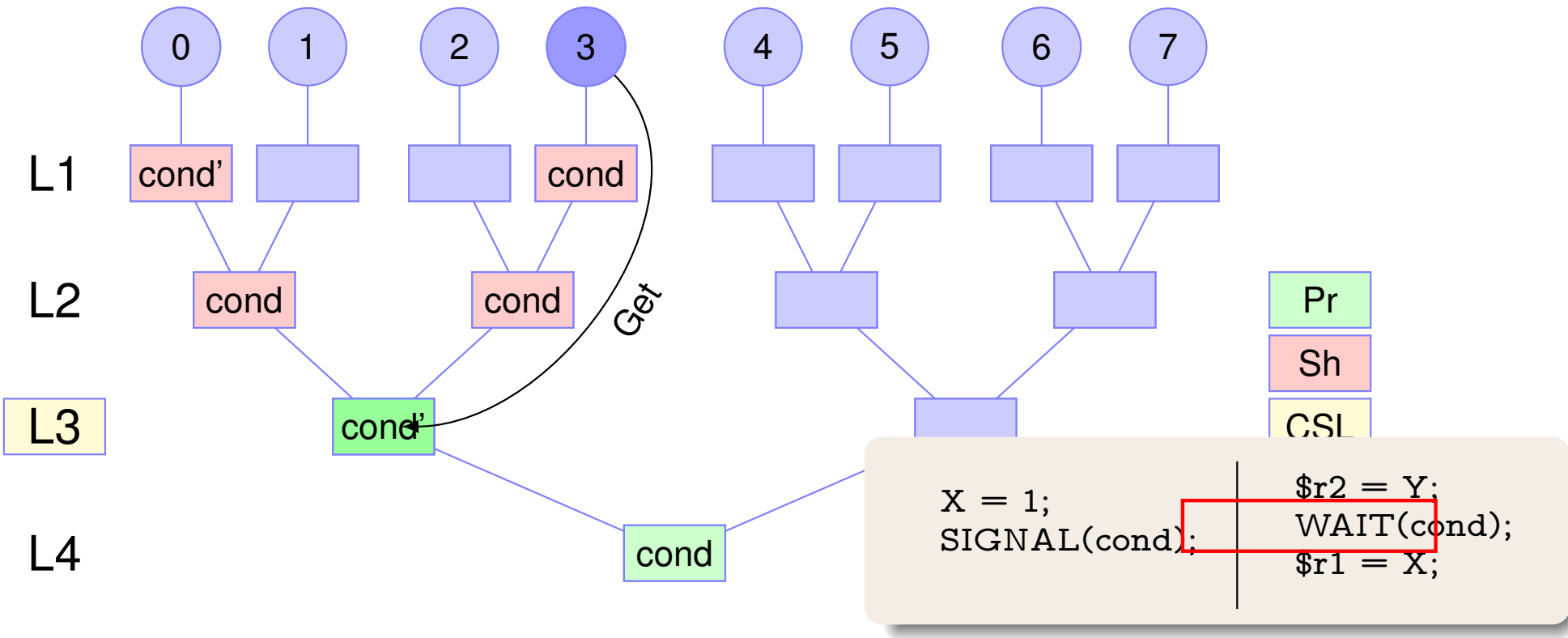
VIPS-H: ATOMIC OPERATIONS

Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
Atomic instr	0.0026%	0.0048%	Low	Low	Medium



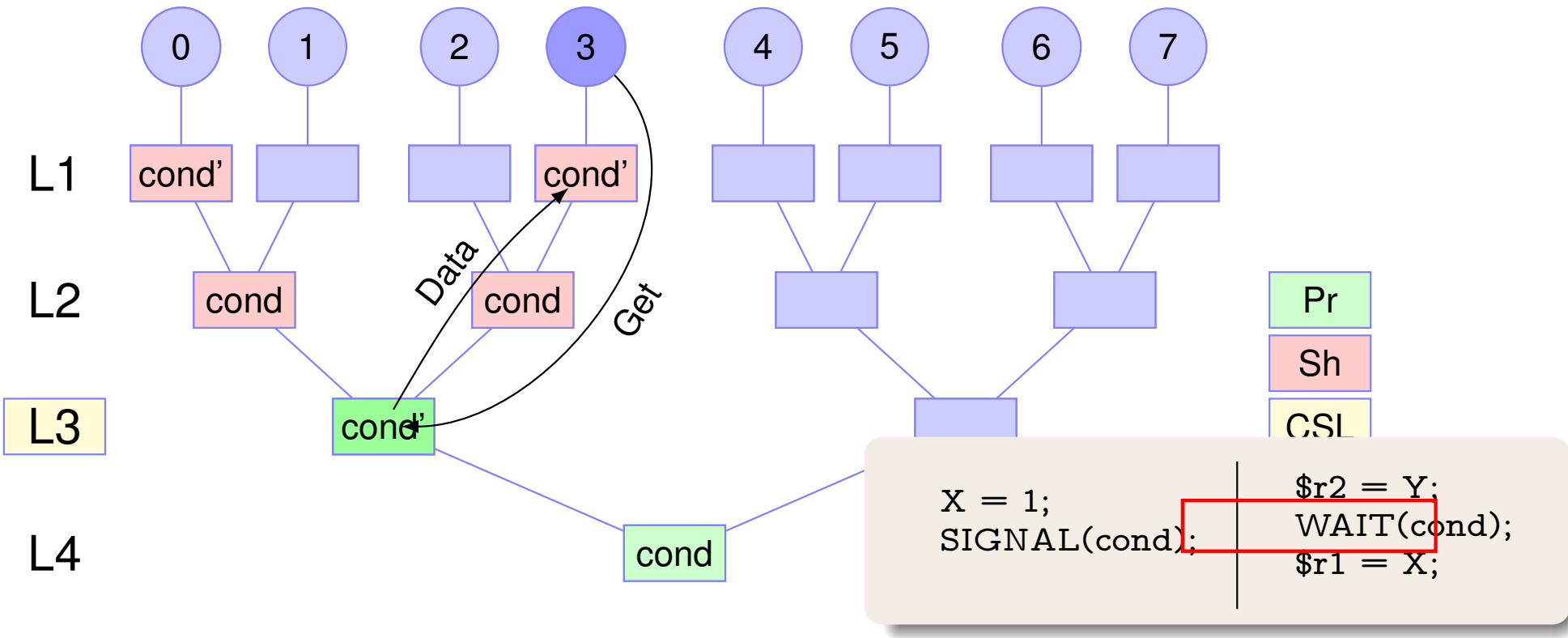
VIPS-H: ATOMIC OPERATIONS

Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
Atomic instr	0.0026%	0.0048%	Low	Low	Medium



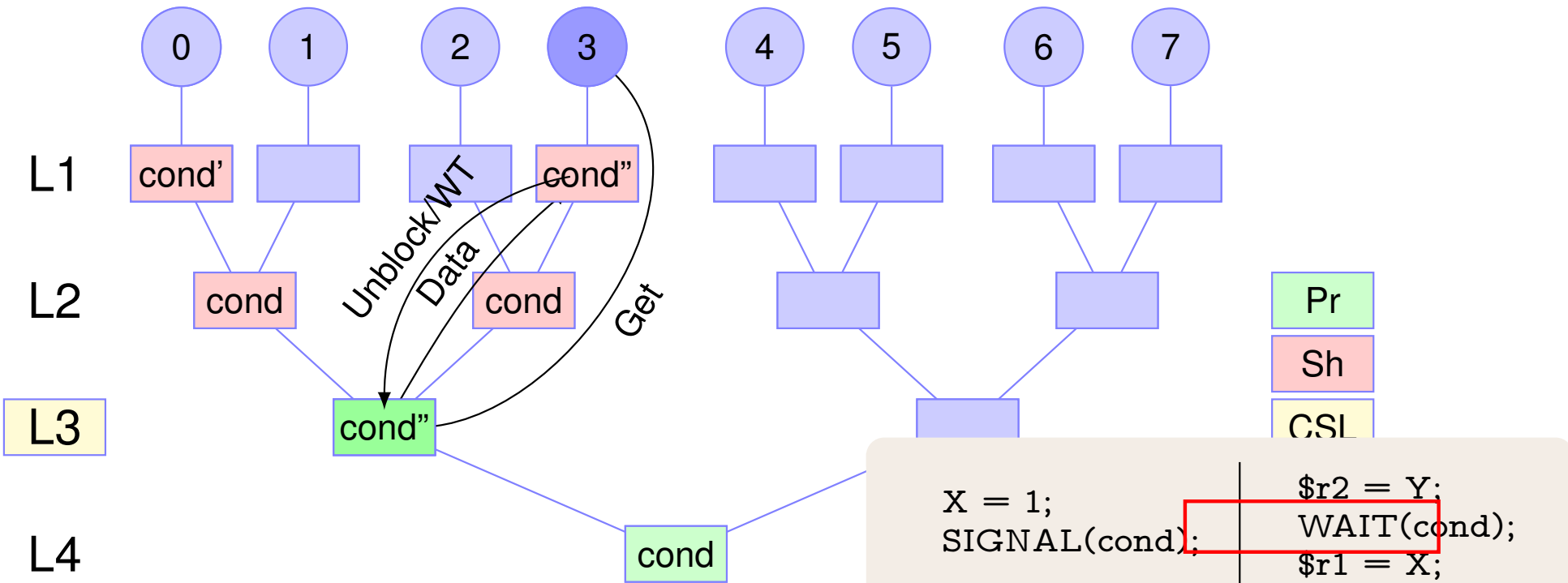
VIPS-H: ATOMIC OPERATIONS

Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
Atomic instr	0.0026%	0.0048%	Low	Low	Medium



VIPS-H: ATOMIC OPERATIONS

Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
Atomic instr	0.0026%	0.0048%	Low	Low	Medium



Recoveries: Updating CSL and P/S

Event	% per instr. (4×4)	% per instr. (16×4)	Frequency	Complexity	Efficiency
Recoveries	0.0006%	0.0005%	Very low	High	Low

- Upon a TLB miss, the CSL of the page is recalculated
- If it changes: recovery to update CSL and P/S info
- Procedure:
  - The page table entry is locked
  - All cores in the old cluster are interrupted
    - Their corresponding TLB entry is invalidated
  - One of the cores in the old cluster set the P/S bit of the blocks in the page as shared
    - In all cache levels in [*old_CSL*, *new_CSL*)
  - The core unlocks the page table entry

Simulation environment

- Simulated a 16-core (4x4) and 64-core (16x4) systems
- Three levels of caches
  - L1: instruction and data 32KB 4-way
  - L2: unified 4MB 16-way
  - L3: unified 16MB 32-way
  - Hierarchical network: Both Garnet and Simple
- Benchmarks: SPLASH-2 (14), scientific (2), PARSEC (6)
- Protocols: H-MOESI, H-TOKEN, VIPS-M, VIPS-H
  - VIPS-H-Multilevel: Store shared blocks in all levels
    - ☺ Better cache usage
  - VIPS-H-Two-level: Only store shared blocks in L1s
    - ☺ Simpler self-invalidation (only in L1s)

Complexity, cost, and area

- H-MOESI: Hierarchical full-map
- H-Token: Tokens and owner token
- VIPS-H: P/S bit

Number of states and extra bits required (16x4)

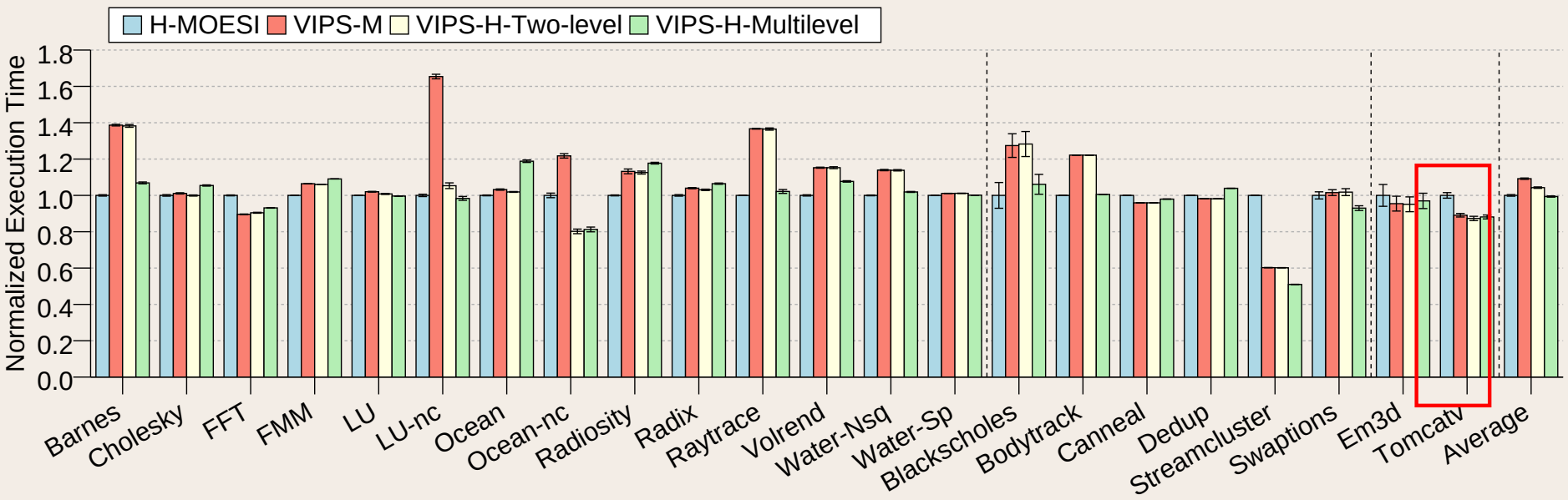
	H-MOESI			H-Token			VIPS-H		
Controller	States Tot./Base	Bitmap bits	Total bits	States Tot./Base	Tokens bits	Total bits	States Tot./Base	P/S bit	Total bits
L1 cache	16 / 5	0	3	16 / 5	7	10	9 / 3	1	3
L2 cache	59 / 13	16	20	6 / 4	7	9	5 / 3	1	3
L3 cache	13 / 4	4	6	3 / 2	7	8	4 / 3	1	3
Total cost	844KB			584KB			204KB		

- 76% memory reduction compared to H-MOESI
- 65% memory reduction compared to H-Token

Comparison to H-MOESI and VIPS-M: Time

- Flat VIPS-M degrades performance by 10% w.r.t. H-MOESI for 4x4, get similar performance for 16x4
- VIPS-H improves execution time by about 11% for 16x4
- VIPS scales better than H-MOESI in time

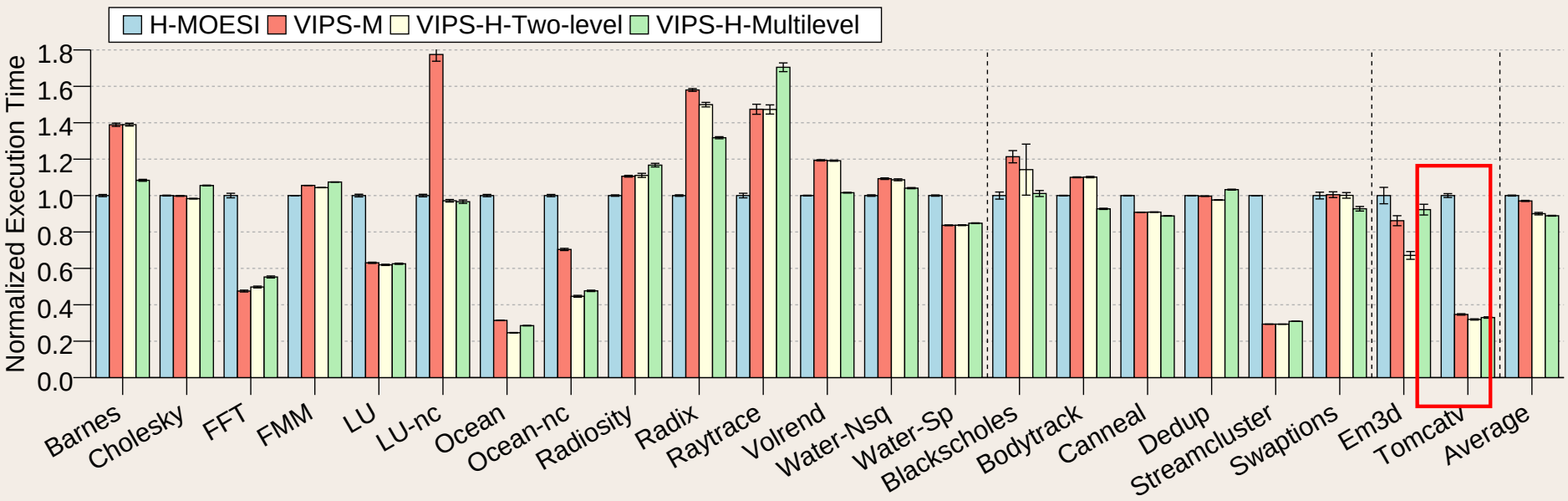
4×4 clustered system



Comparison to H-MOESI and VIPS-M: Time

- Flat VIPS-M degrades performance by 10% w.r.t. H-MOESI for 4x4, get similar performance for 16x4
- VIPS-H improves execution time by about 11% for 16x4
- VIPS scales better than H-MOESI in time

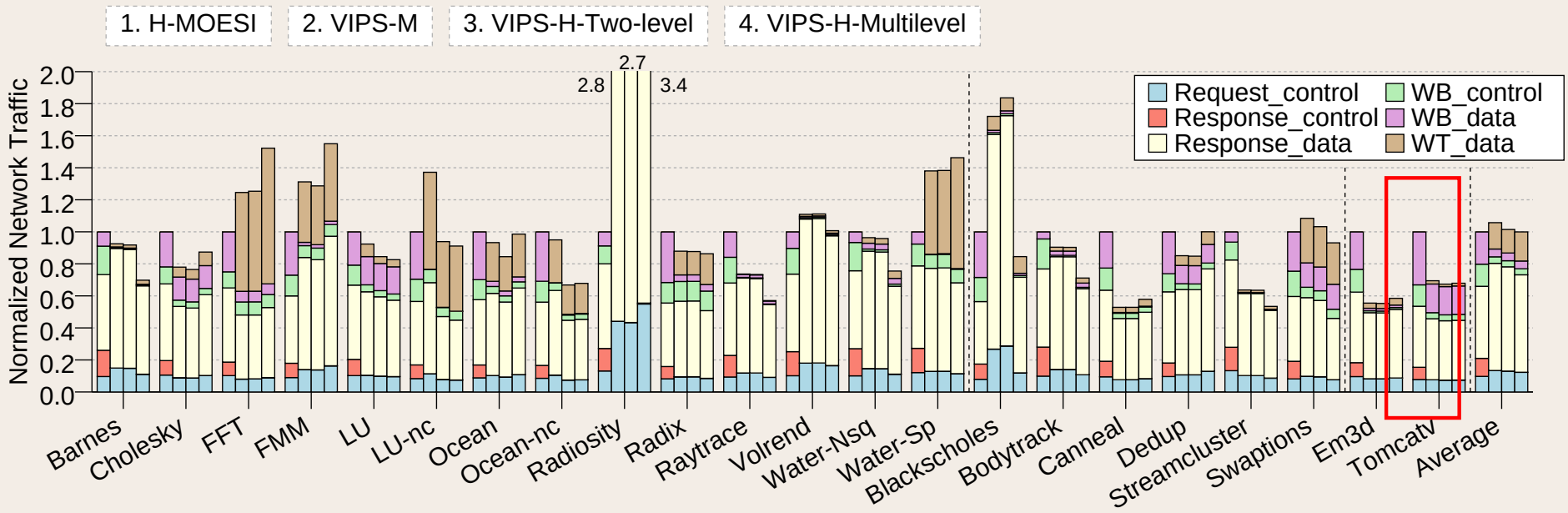
16×4 clustered system



Comparison to H-MOESI and VIPS-M: Traffic

- VIPS increases *Response_data* $\Rightarrow$ more cache misses
- But less control traffic $\Rightarrow$ no invalidations acks
- It scales better than H-MOESI in traffic (5%–7% for 16x4)

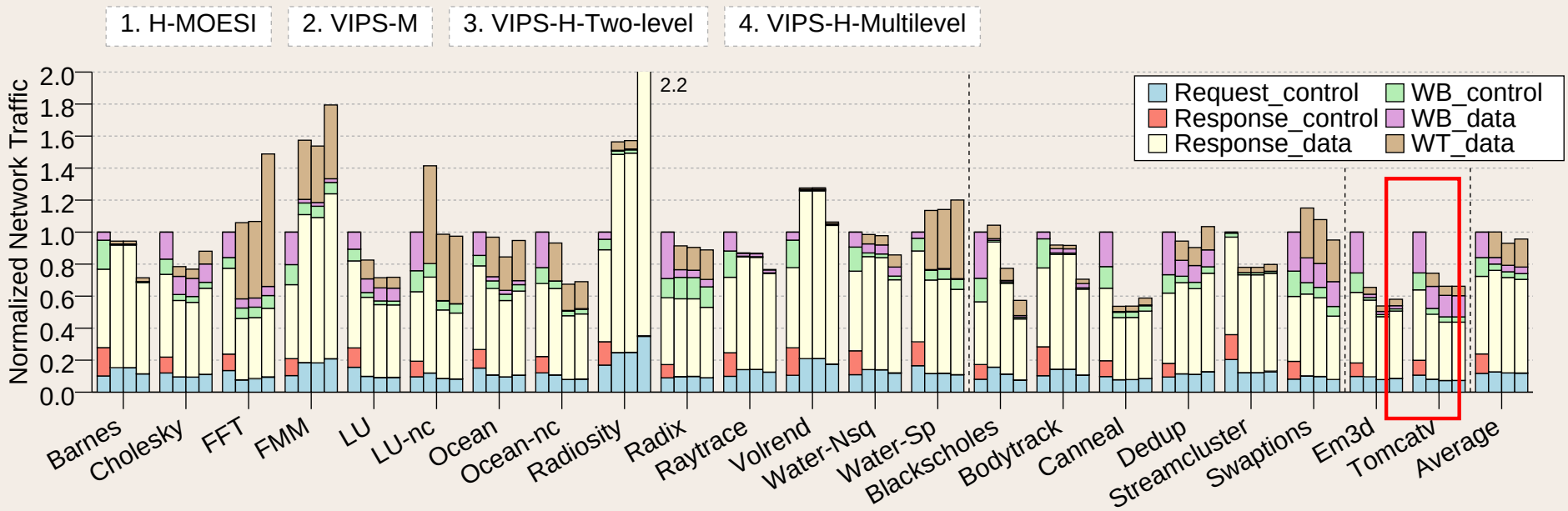
4×4 clustered system



Comparison to H-MOESI and VIPS-M: Traffic

- VIPS increases *Response_data* $\Rightarrow$ more cache misses
- But less control traffic $\Rightarrow$ no invalidations acks
- It scales better than H-MOESI in traffic (5%–7% for 16x4)

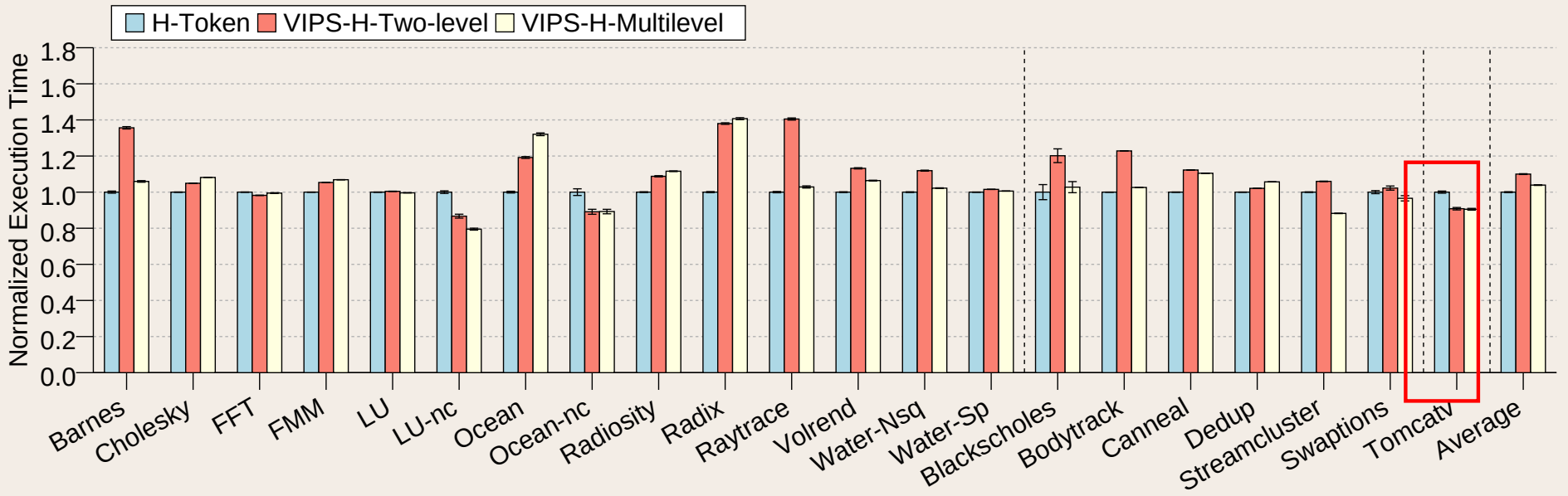
16×4 clustered system



Comparison to H-Token: Execution time

- H-Token achieves better performance than VIPS-H with an ideal network
- The performance figures are closer for the 16x4 configuration

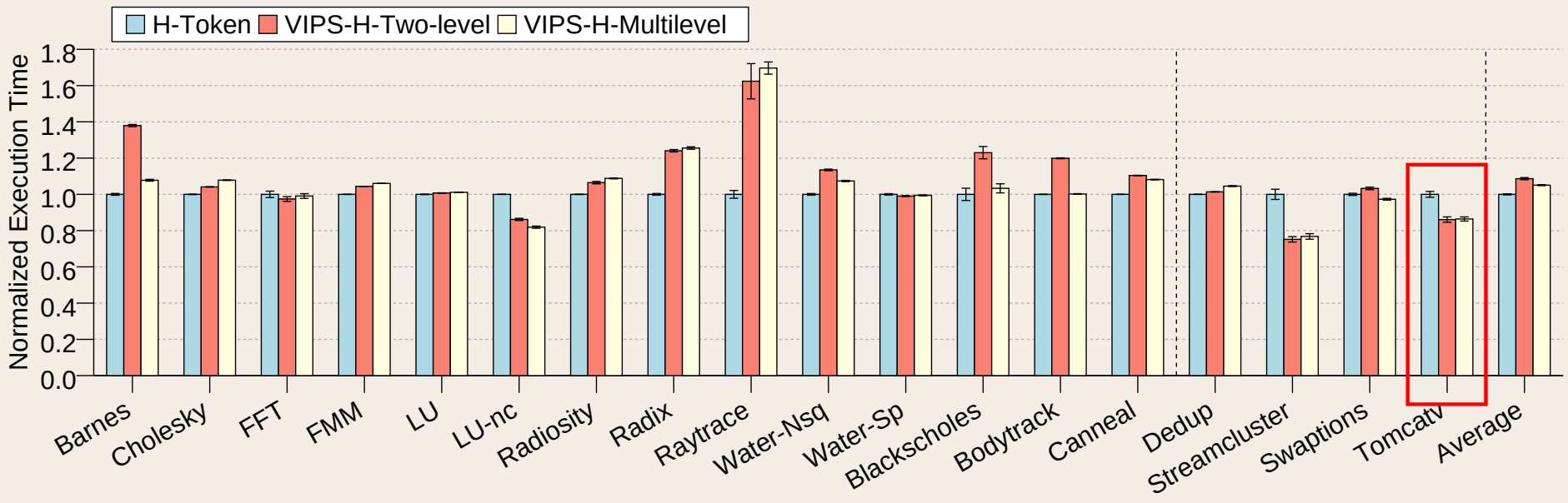
4×4 clustered system



Comparison to H-Token: Execution time

- H-Token achieves better performance than VIPS-H with an ideal network
- The performance figures are closer for the 16x4 configuration

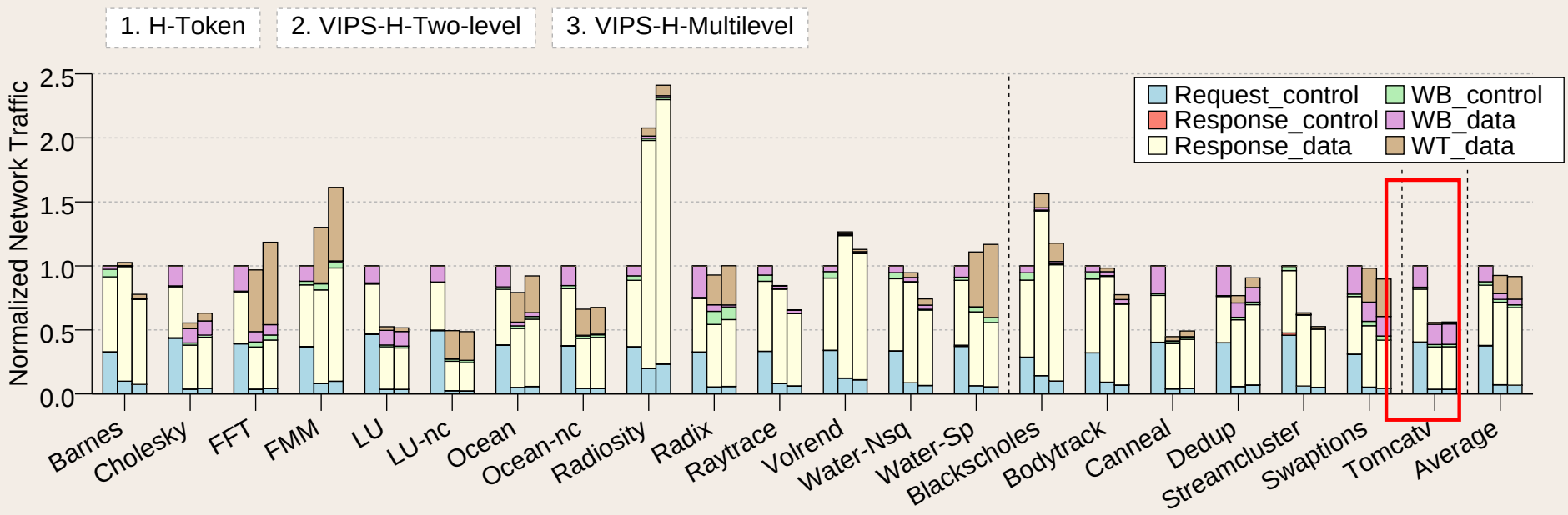
16×4 clustered system



Comparison to H-Token: Network traffic

- H-Token requires more network traffic that VIPS-H for 4x4
- It increases dramatically for the 16x4 configuration

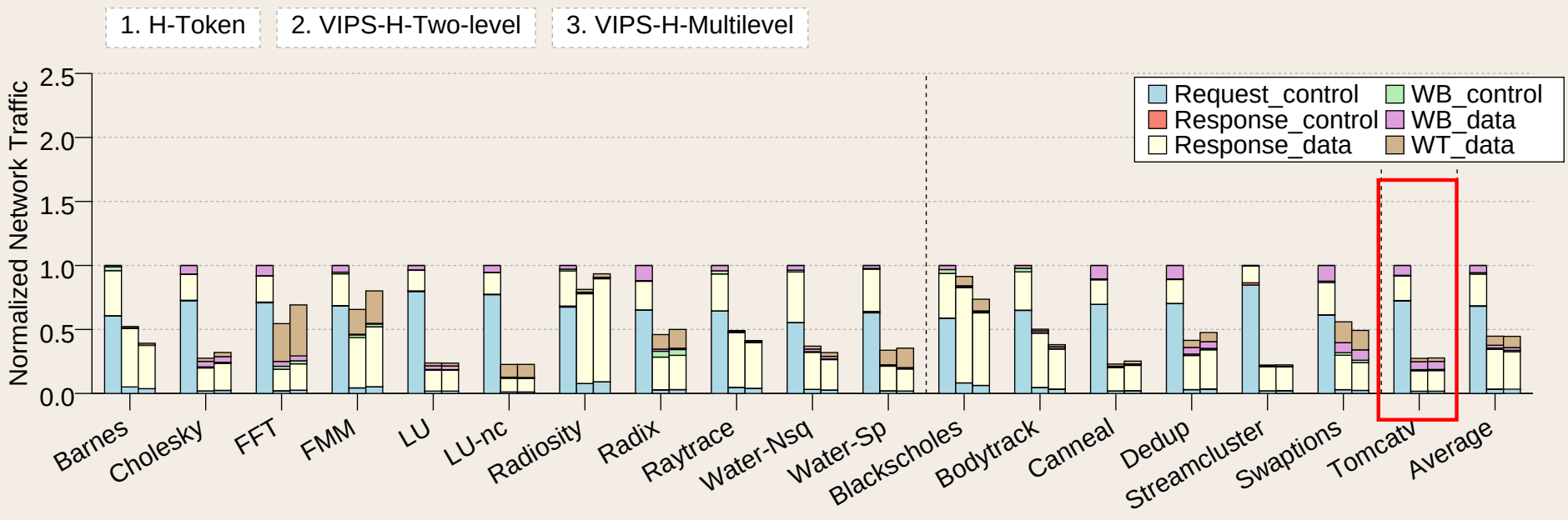
4×4 clustered system



Comparison to H-Token: Network traffic

- H-Token requires more network traffic that VIPS-H for 4x4
- It increases dramatically for the 16x4 configuration

16×4 clustered system



Conclusions

- Simple and efficient cache coherence for clustered cache architectures
- Keys:
  - Self-invalidation and self-downgrade and the assumption of SC for DRF semantics
  - Hierarchical private/shared classification
- Results:
  - Simpler than H-MOESI
    - Less states memory overhead (from 94 to 18)
    - Less memory overhead (76%)
  - Better performance (11%, on average for 16x4)
  - Reduced network traffic (7%, on average for 16x4)
  - Better scalability

Motivation

- Need of simple, scalable, and efficient cache coherence
  - Many-core systems, GPUs?, accelerators??

Motivation

- Need of simple, scalable, and efficient cache coherence
  - Many-core systems, GPUs?, accelerators??
- Traditional directory protocols
  - Explicit invalidation/downgrades on writes/reads
    $\Rightarrow$ Complex
  - Directory to track copies $\Rightarrow$ Non-scalable
  - Indirection $\Rightarrow$ Inefficient

SIMPLE CACHE COHERENCE PROTOCOLS

- Simple cache coherence: VIPS-M *[Ros & Kaxiras, PACT'12]*
  - Strictly request-response ⇒ Simple
  - Coherence distributed across cores ⇒ Scalable
  - No directory ⇒ Simple and scalable
  - Efficient for data-race-free code

SIMPLE CACHE COHERENCE PROTOCOLS

- Simple cache coherence: VIPS-M *[Ros & Kaxiras, PACT'12]*
  - Strictly request-response ⇒ Simple
  - Coherence distributed across cores ⇒ Scalable
  - No directory ⇒ Simple and scalable
  - Efficient for data-race-free code
- How? Self-invalidation and self-downgrade (SISD)
  - Synchronization exposed to the protocol

EXAMPLE OF DRF CODE

```
/* Initially X = 0 */

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

SIMPLE CACHE COHERENCE PROTOCOLS

- Simple cache coherence: VIPS-M *[Ros & Kaxiras, PACT'12]*
  - Strictly request-response ⇒ Simple
  - Coherence distributed across cores ⇒ Scalable
  - No directory ⇒ Simple and scalable
  - Efficient for data-race-free code
- How? Self-invalidation and self-downgrade (SISD)
  - Synchronization exposed to the protocol
- Release: Self-downgrade
  - ⇒ Write-through dirty blocks

SD

EXAMPLE OF DRF CODE

```
/* Initially X = 0 */

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

SIMPLE CACHE COHERENCE PROTOCOLS

- Simple cache coherence: VIPS-M *[Ros & Kaxiras, PACT'12]*
  - Strictly request-response ⇒ Simple
  - Coherence distributed across cores ⇒ Scalable
  - No directory ⇒ Simple and scalable
  - Efficient for data-race-free code
- How? Self-invalidation and self-downgrade (SISD)
  - Synchronization exposed to the protocol
- Release: Self-downgrade
  - ⇒ Write-through dirty blocks
- Acquire: Self-invalidation
  - ⇒ Empty the cache

EXAMPLE OF DRF CODE

```
/* Initially X = 0 */

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

SI

SIMPLE CACHE COHERENCE PROTOCOLS

- Simple cache coherence: VIPS-M *[Ros & Kaxiras, PACT'12]*
  - Strictly request-response ⇒ Simple
  - Coherence distributed across cores ⇒ Scalable
  - No directory ⇒ Simple and scalable
  - Efficient for data-race-free code
- How? Self-invalidation and self-downgrade (SISD)
  - Synchronization exposed to the protocol
- Release: Self-downgrade
  - ⇒ Write-through dirty blocks
- Acquire: Self-invalidation
  - ⇒ Empty the cache

EXAMPLE OF DRF CODE

```
/* Initially X = 0 */

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

- Sequential consistency (SC) for data-race-free (DRF) code

THE PROBLEM OF THE DATA RACES

- Even DRF applications contain races!
  - Synchronization is inherently racy

EXAMPLE OF DRF CODE

```
/* Initially X = 0 */

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

THE PROBLEM OF THE DATA RACES

- Even DRF applications contain races!
  - Synchronization is inherently racy
  - Implemented performing spin-waiting

EXAMPLE OF DRF CODE

```
/* Initially X = 0 */

X = 1;                    |  while(!cond);
cond = 1;                 |  $r1 = X;
```

THE PROBLEM OF THE DATA RACES

- Even DRF applications contain races!
  - Synchronization is inherently racy
  - Implemented performing spin-waiting
- Spin-waiting is not efficient under SISD
  - Writes require fast propagation
    - Write-through and repeated self-invalidation

EXAMPLE OF DRF CODE

```
/* Initially X = 0 */

X = 1;          |  while(!cond);
cond = 1;       |  $r1 = X;
```

THE PROBLEM OF THE DATA RACES

- Even DRF applications contain races!
  - Synchronization is inherently racy
  - Implemented performing spin-waiting
- Spin-waiting is not efficient under SISD
  - Writes require fast propagation
    - Write-through and repeated self-invalidation
  - Repeated self-invalidation ⇒ spin on last level cache (LLC)
    - Increases network traffic and LLC accesses ⇒ energy

EXAMPLE OF DRF CODE

```
/* Initially X = 0 */

X = 1;
cond = 1;
```

```
while(!cond);
$r1 = X;
```

SI

THE PROBLEM OF THE DATA RACES

- Even DRF applications contain races!
  - Synchronization is inherently racy
  - Implemented performing spin-waiting
- Spin-waiting is not efficient under SISD
  - Writes require fast propagation
    - Write-through and repeated self-invalidation
  - Repeated self-invalidation ⇒ spin on last level cache (LLC)
    - Increases network traffic and LLC accesses ⇒ energy
- VIPS-M solution
  ⇒ Exponential back-off
  - ☺ Reduces SI, network traffic, and LLC accesses
  - ☹ Slows down propagation of writes

EXAMPLE OF DRF CODE

```
/* Initially X = 0 */

X = 1;
cond = 1;

                    SI
while(!cond);
$r1 = X;
```

THE PROBLEM OF THE DATA RACES

- Even DRF applications contain races!
  - Synchronization is inherently racy
  - Implemented performing spin-waiting
- Spin-waiting is not efficient under SISD
  - Writes require fast propagation
    - Write-through and repeated self-invalidation
  - Repeated self-invalidation ⇒ spin on last level cache (LLC)
    - Increases network traffic and LLC accesses ⇒ energy
- VIPS-M solution
  ⇒ Exponential back-off
  - ☺ Reduces SI, network traffic, and LLC accesses
  - ☹ Slows down propagation of writes

EXAMPLE OF DRF CODE

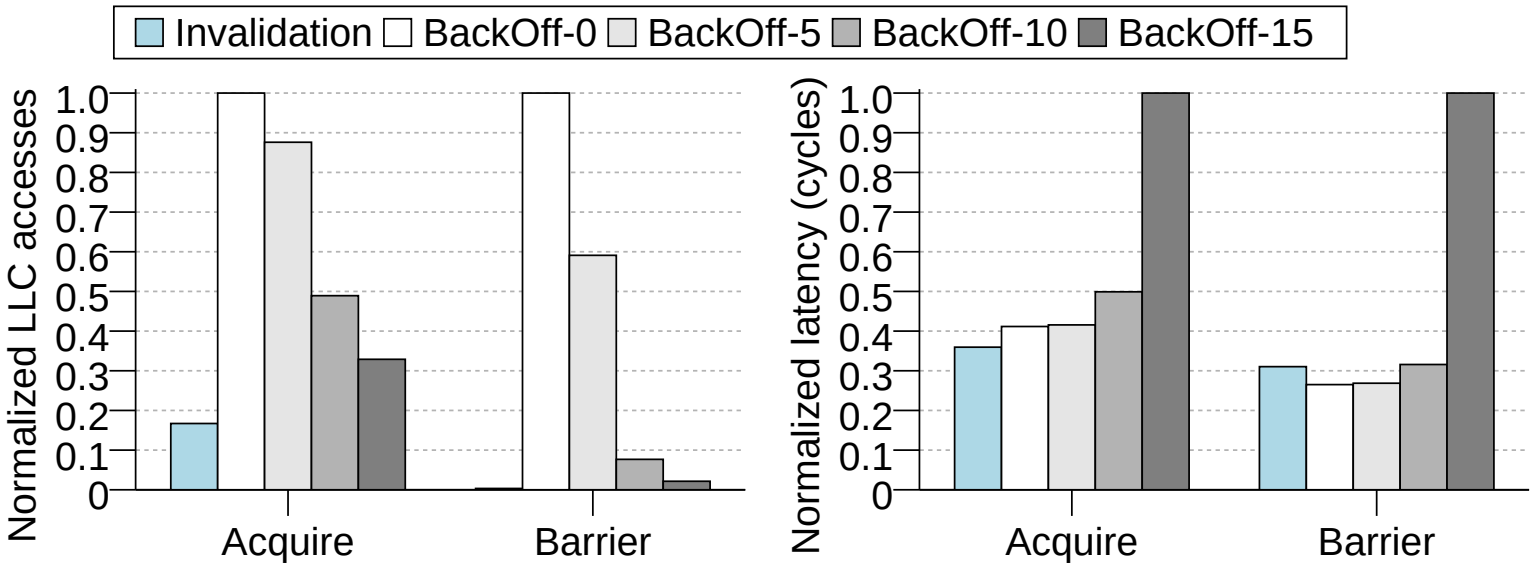
```
/* Initially X = 0 */

X = 1;                  | while(!cond);
cond = 1;               | $r1 = X;
```

SI

Energy-performance trade-off!

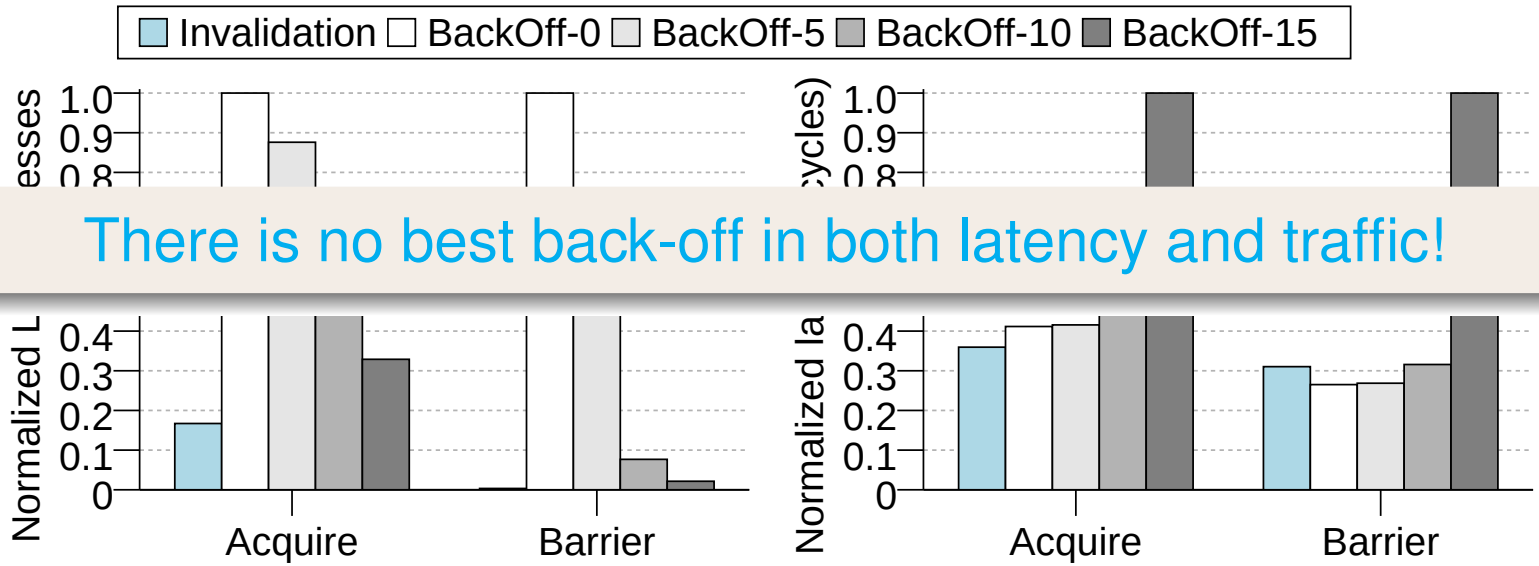
ENERGY-PERFORMANCE TRADE-OFF



Traffic

Latency

ENERGY-PERFORMANCE TRADE-OFF

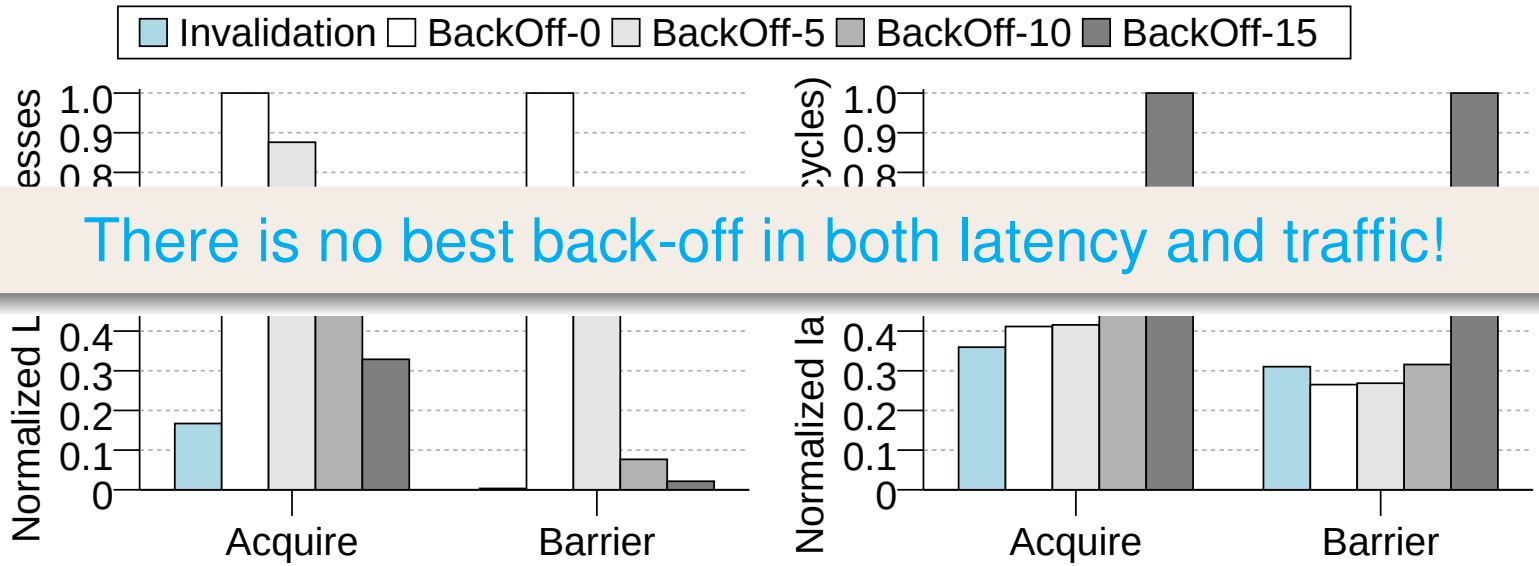


There is no best back-off in both latency and traffic!

Traffic

Latency

ENERGY-PERFORMANCE TRADE-OFF



There is no best back-off in both latency and traffic!

THE CHALLENGE

Fast and efficient write propagation...

- without explicit invalidations/downgrades
- keeping a simple request-response protocol

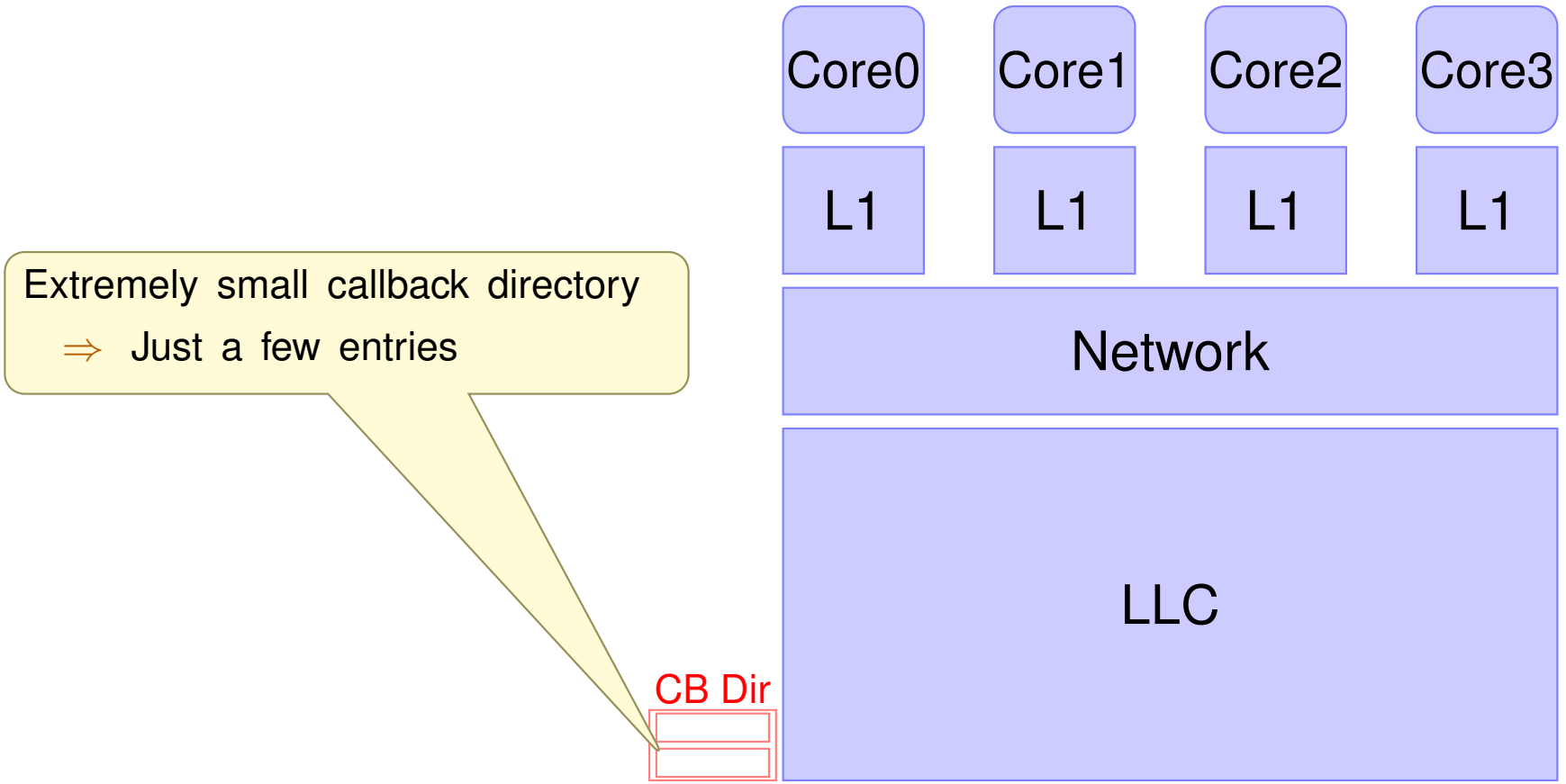
CALLBACKS

- A mechanism with a directory just for races involved in spin-waiting
  - Only special loads (or atomics) called LOAD_CALLBACK (LD_CB) can allocate an entry in the directory

CALLBACKS

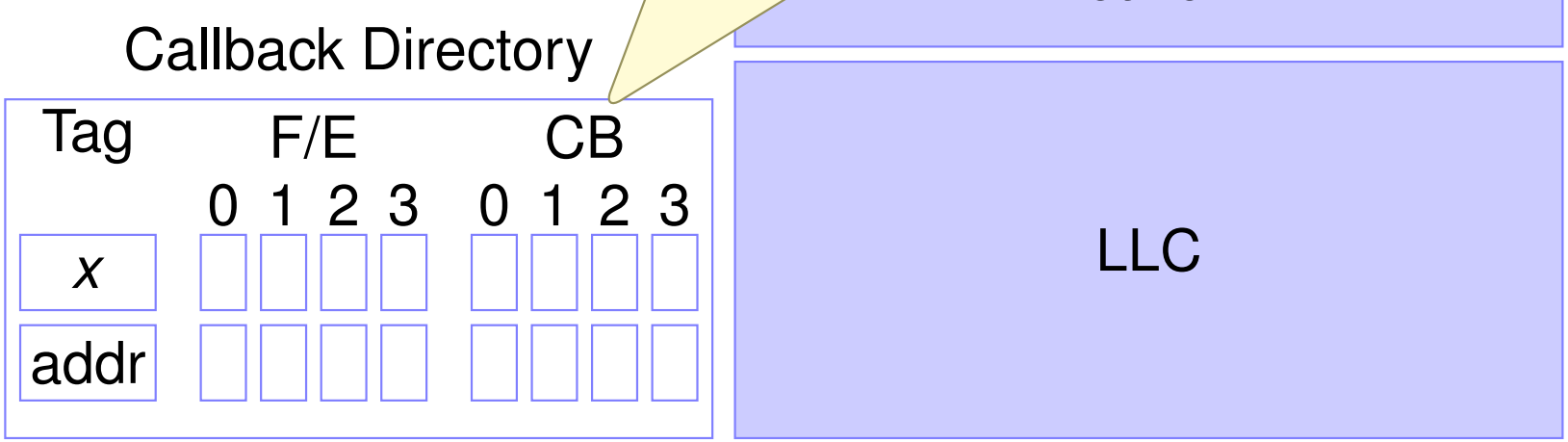
- A mechanism with a directory just for races involved in spin-waiting
  - Only special loads (or atomics) called LOAD_CALLBACK (LD_CB) can allocate an entry in the directory
- A LD_CB is similar to a load instruction, but
  - By-passes the private caches
  - May block at the shared cache waiting for a write to happen

CALLBACK EXAMPLE

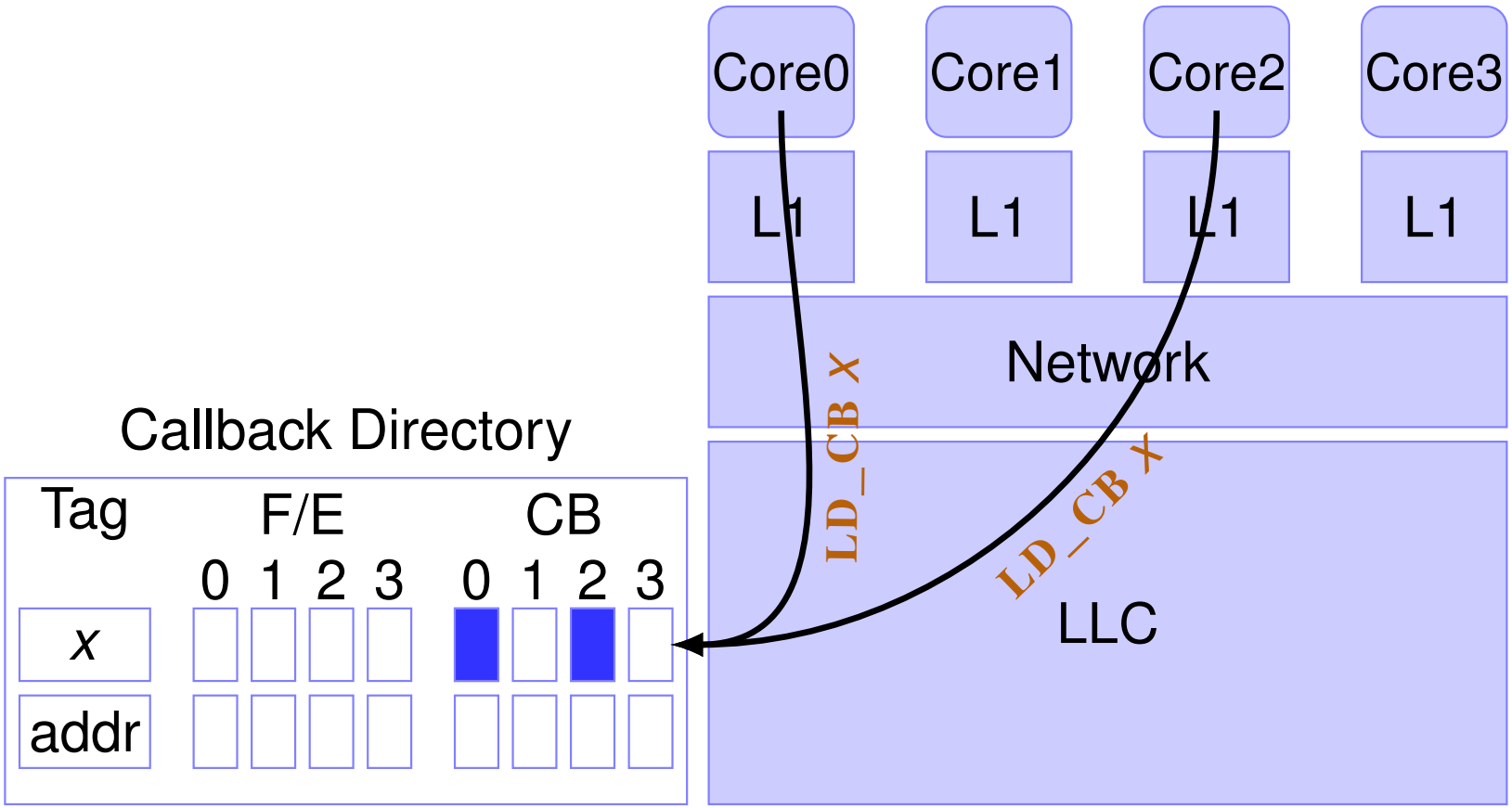


CALLBACK EXAMPLE

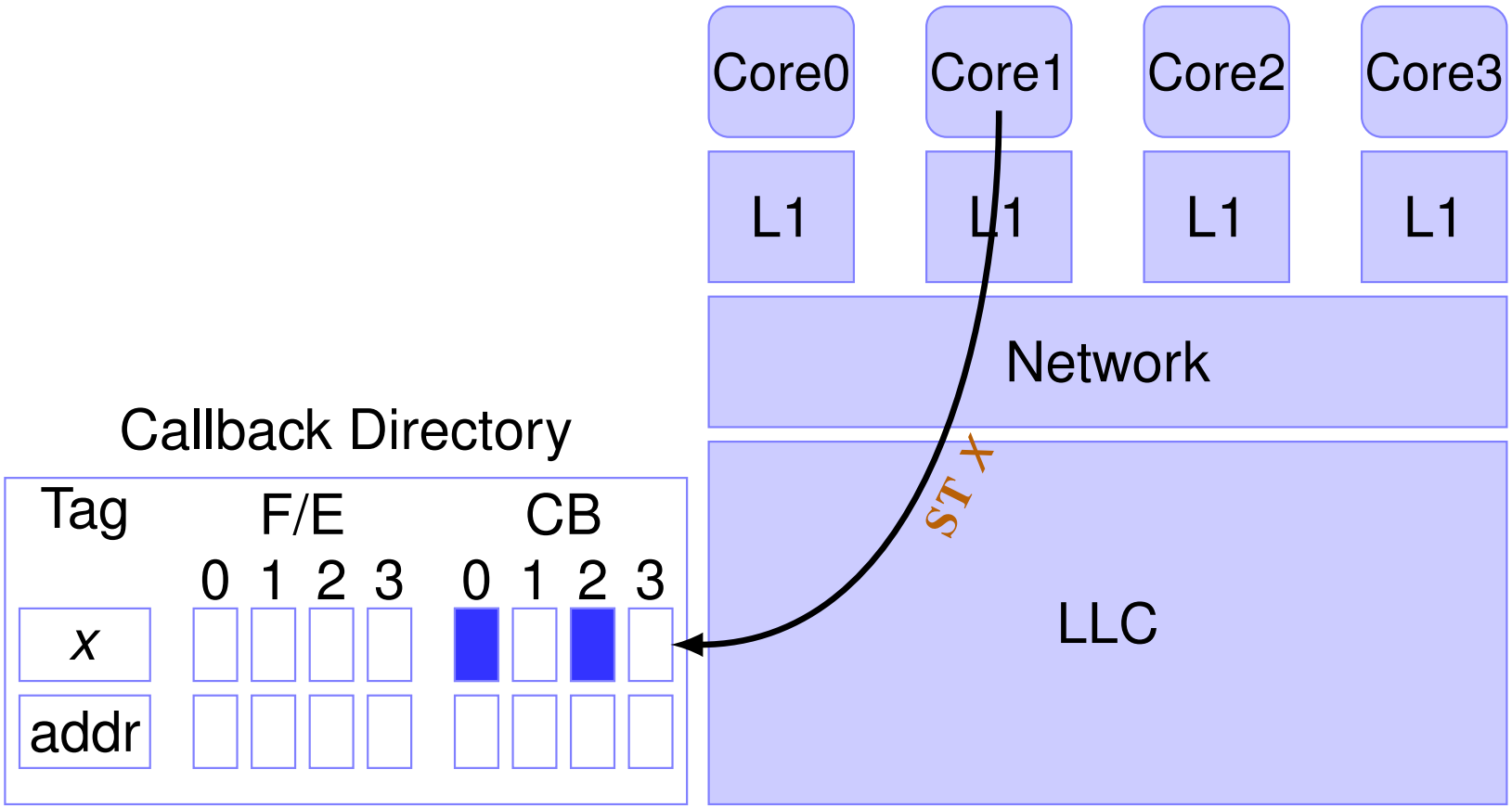
- Tag: word address
- F/E: Full/Empty bits per core
  - Full: It may be a new value to consume
  - Empty: There is no new value
- CB: Callback bits per core
  - A callback is blocked waiting for a new value



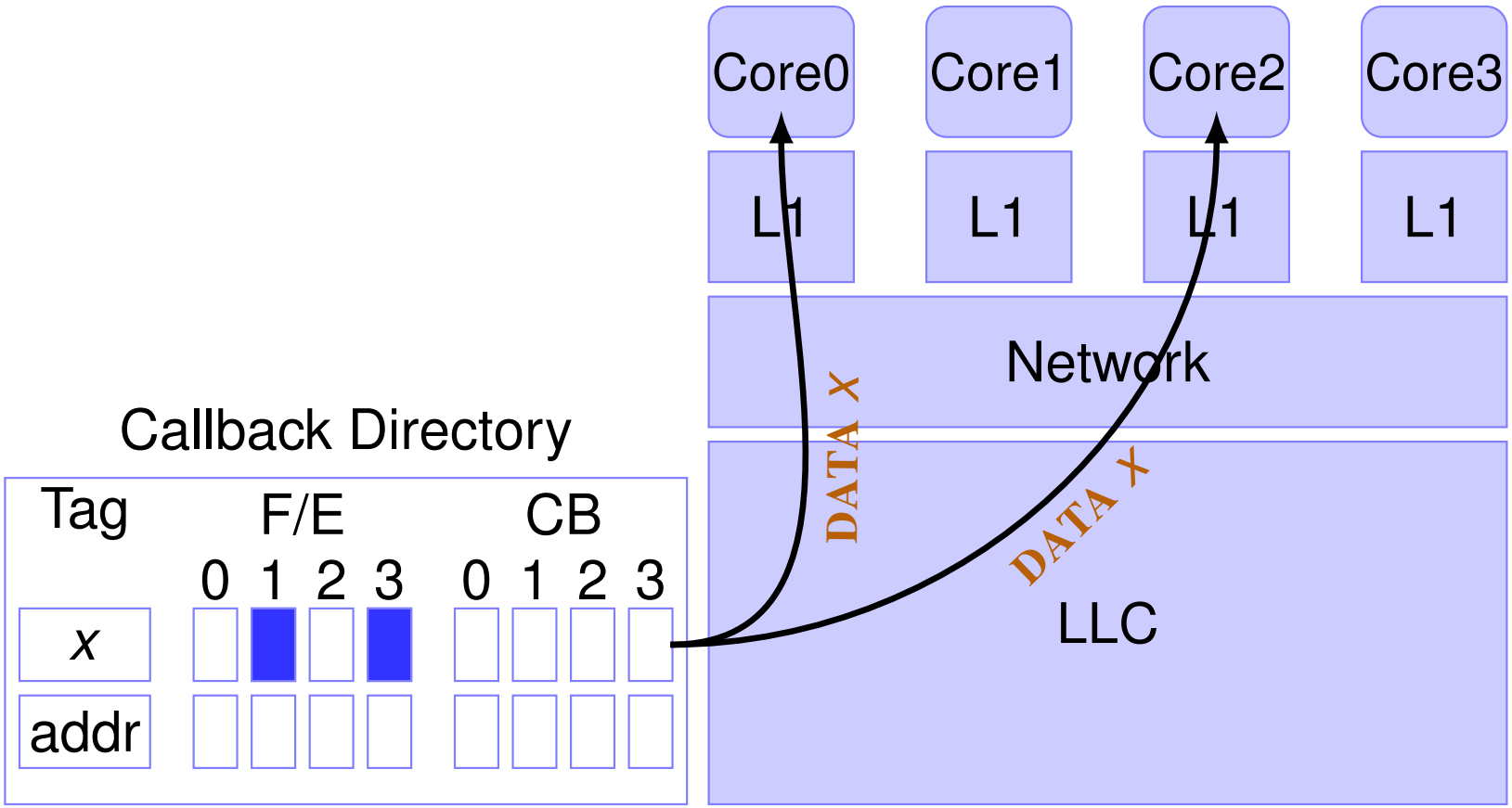
CALLBACK EXAMPLE



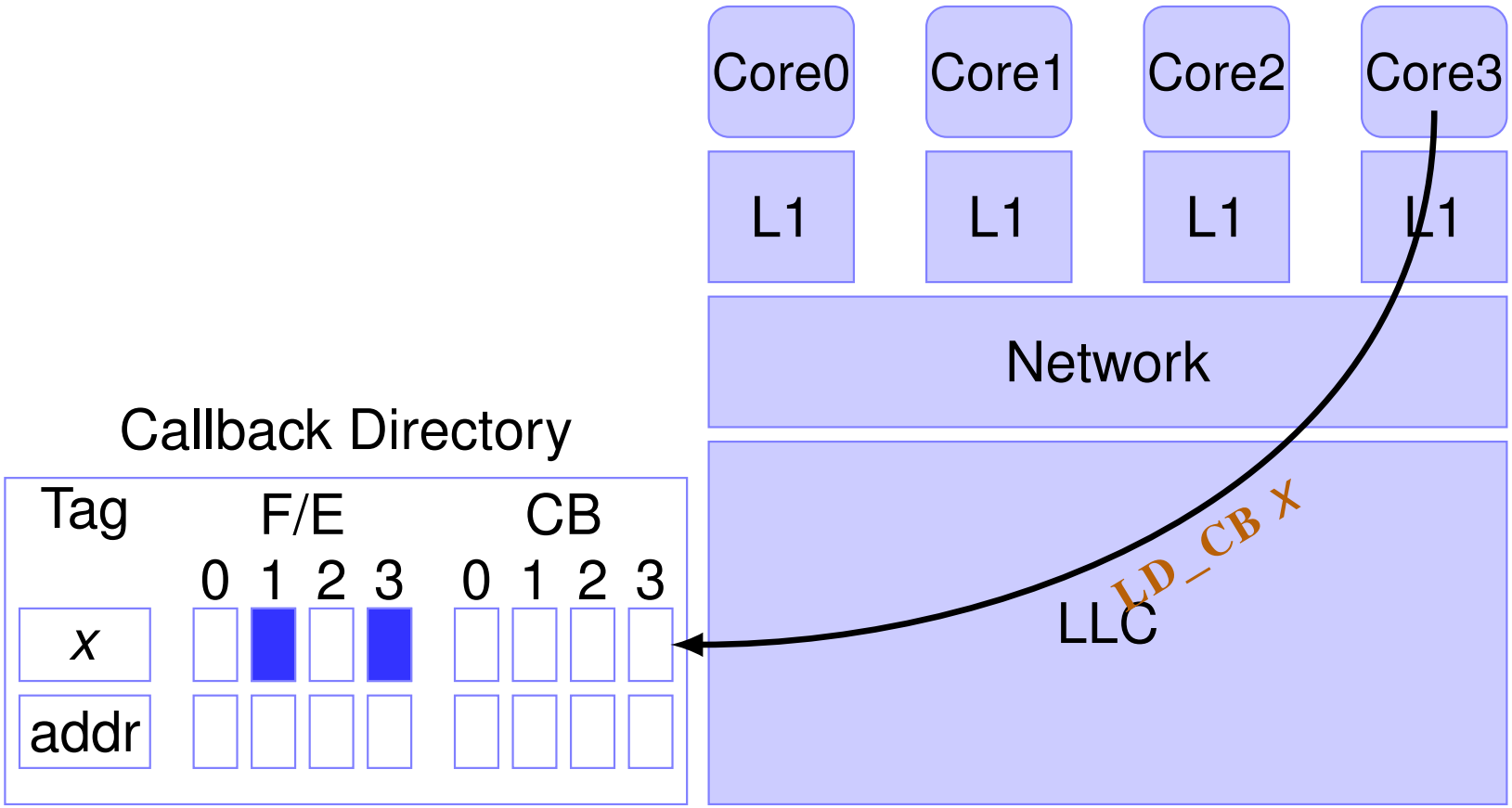
CALLBACK EXAMPLE



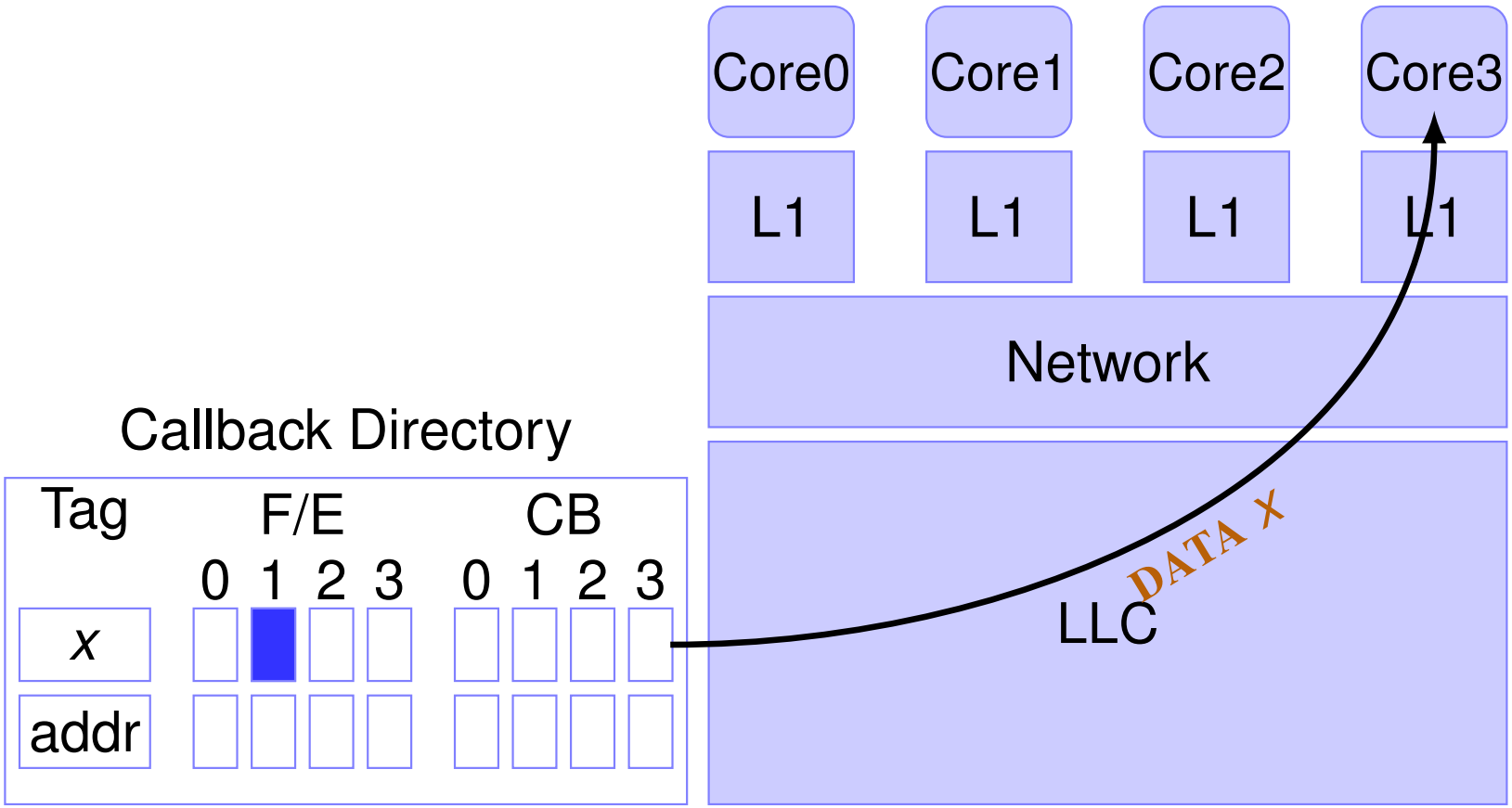
CALLBACK EXAMPLE



CALLBACK EXAMPLE



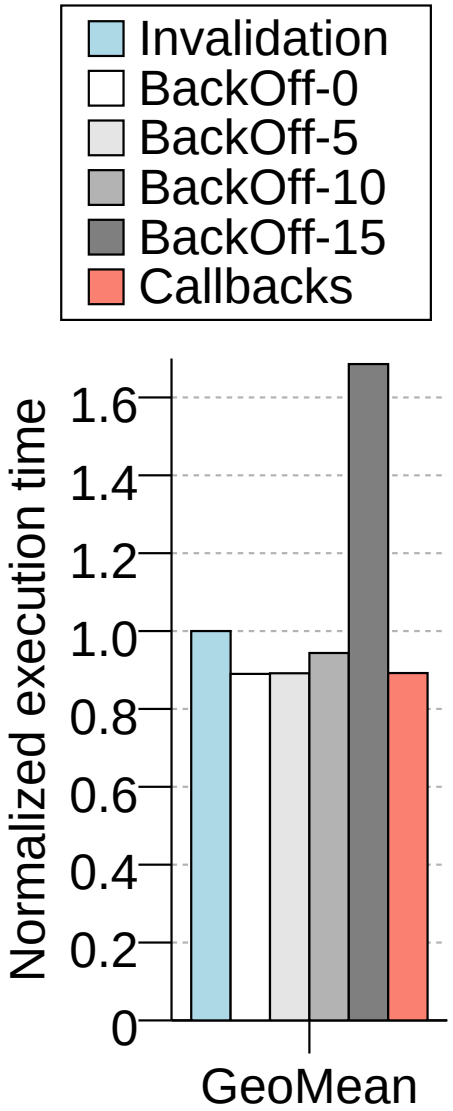
CALLBACK EXAMPLE



EXECUTION TIME AND ENERGY CONSUMPTION

Execution time

- As good as the best BACKOFF case
- 5% better than BACKOFF-10 (VIPS-M)
- 11% better than INVALIDATION



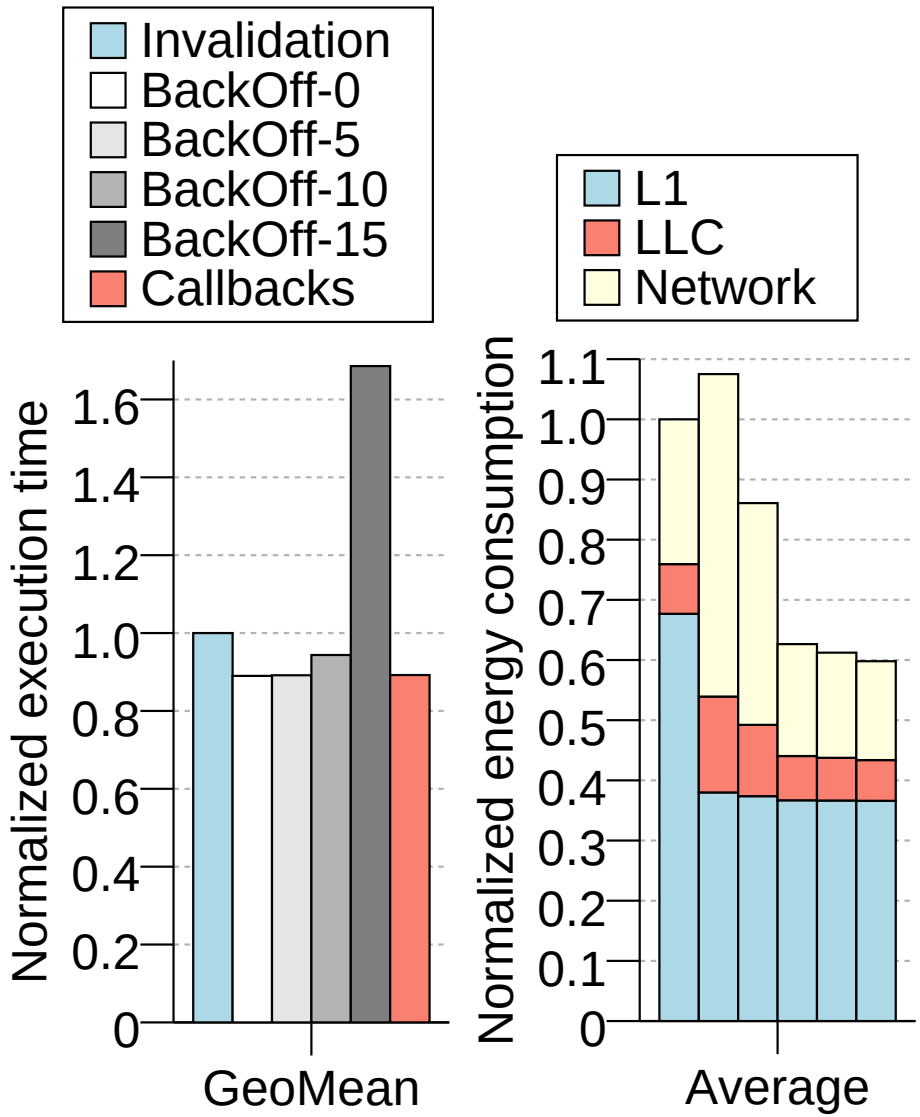
EXECUTION TIME AND ENERGY CONSUMPTION

Execution time

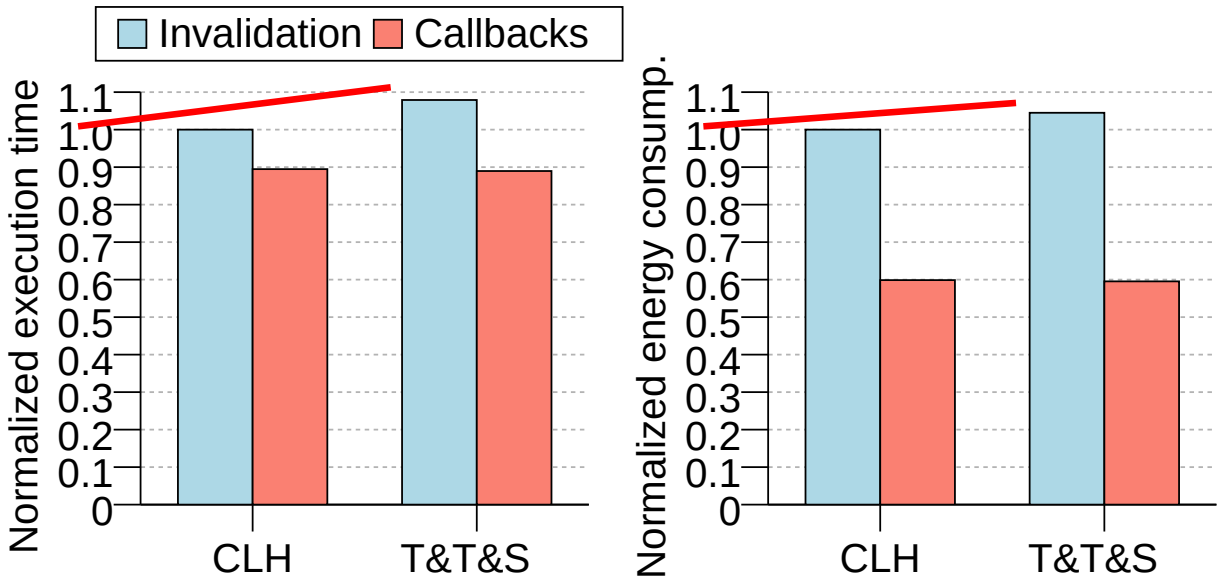
- As good as the best BACKOFF case
- 5% better than BACKOFF-10 (VIPS-M)
- 11% better than INVALIDATION

Energy consumption

- INVALIDATION spins in L1
- BACKOFF-0 spins in the LLC
- CALLBACKS removes spinning (40% and 5% reduction)

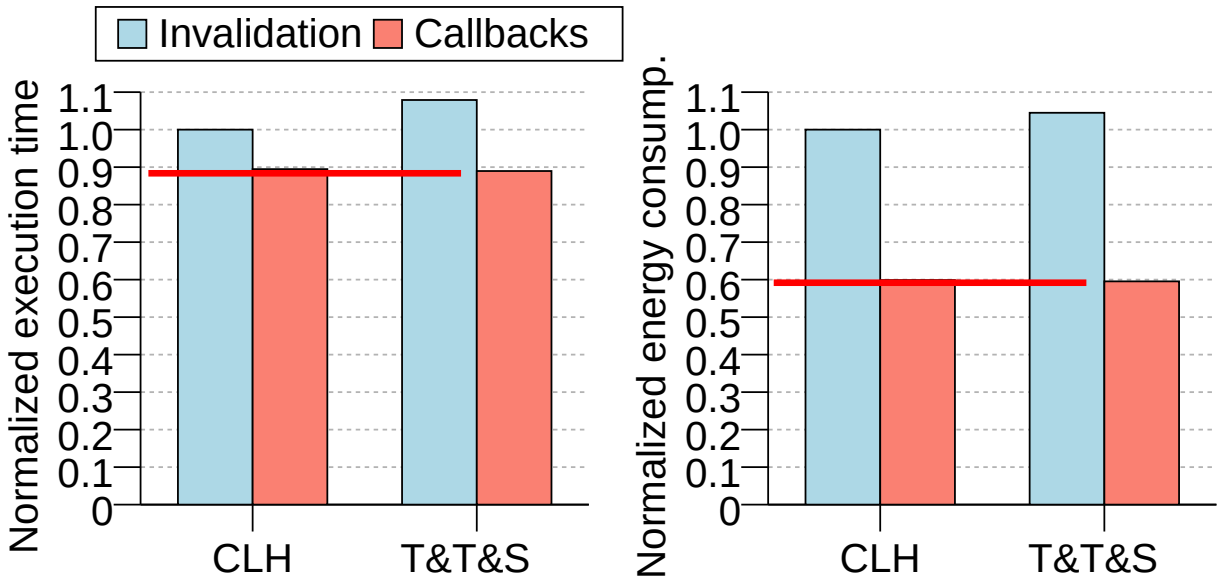


TATAS vs. CLH


Invalidation
Callbacks
Normalized execution time
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
CLH
T&T&S
Normalized energy consump.
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
CLH
T&T&S

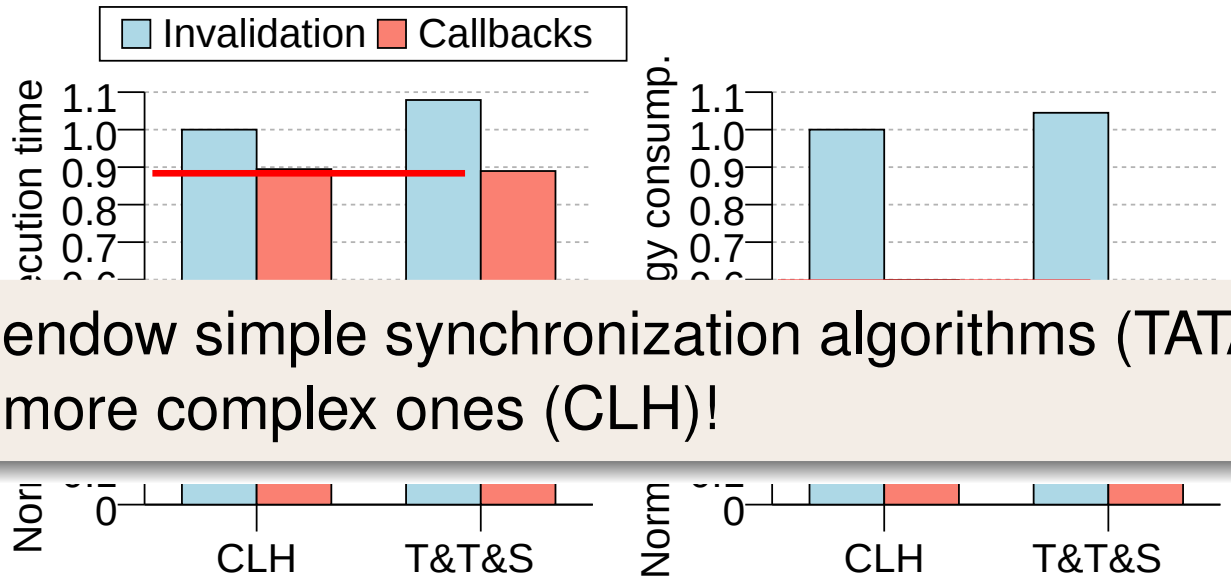
TATAS vs. CLH

- T&T&S + Callbacks allows only one of the threads to race for acquiring the lock
- T&T&S + Callbacks provides fairness



TATAS vs. CLH

- T&T&S + Callbacks allows only one of the threads to race for acquiring the lock
- T&T&S + Callbacks provides fairness



CALLBACKS endow simple synchronization algorithms (TATAS) with the efficiency of more complex ones (CLH)!

TAKE AWAY MESSAGE

- CALLBACKS: special loads for races in spin-waiting
  - $\Rightarrow$ Requires a very small directory
- Simpler and more efficient than explicit invalidation!
- Transparent to the coherence protocol
- Makes efficient simple synchronization algorithms, such as T&T&S

ARGO

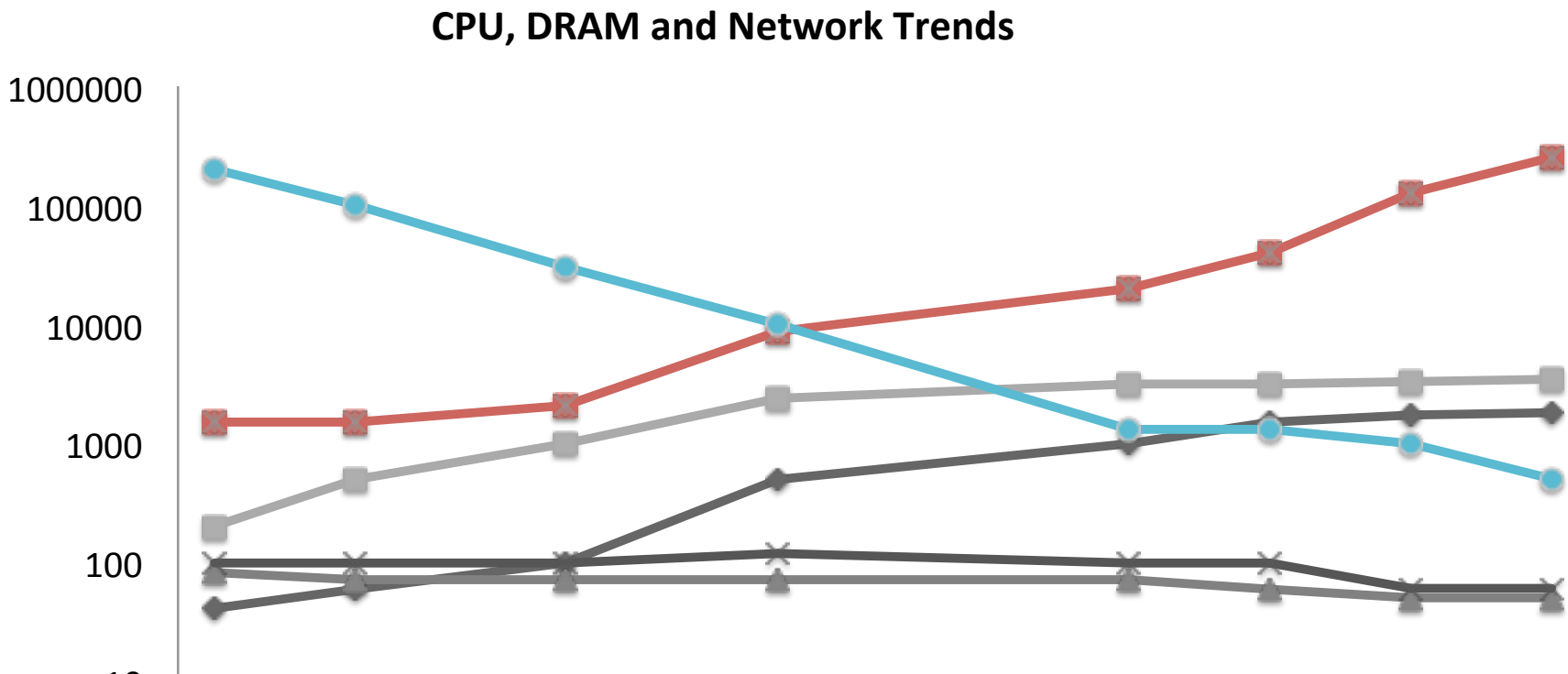
- VIPS-M $\Rightarrow$ Self-Invalidation & Self-Downgrade
  - VIPS coherence is truly distributed.
  - Coherence decisions are taken independently without any inter-core interaction
    - $\Rightarrow$ Simplifies whole system design
- Request-Response from the L1s to the LLC
  - No requests from LCC to L1s
  - No traffic among L1s, only L1 $\Leftrightarrow$ LLC

Argo

- VIPS-M $\Rightarrow$ Self-Invalidation & Self-Downgrade
  - VIPS coherence is truly distributed.
  - Coherence decisions are taken independently without any inter-core interaction
    - $\Rightarrow$ Simplifies whole system design
- Request-Response from the L1s to the LLC
  - No requests from LCC to L1s
  - No traffic among L1s, only L1 $\Leftrightarrow$ LLC

Can this be the answer to distributed coherence?

TRENDS: WHY NOW?



	1992	1994	1997	2000	2005	2007	2009	2011
CPU speed High (MHz)	40	60	100	500	1000	1500	1700	1800
CPU speed Low (MHz)	200	500	1000	2400	3200	3200	3300	3400
DRAM Latency Low (ns)	80	70	70	70	70	60	50	50
DRAM Latency High (ns)	100	100	100	120	100	100	60	60
Network BW (Mbps)	1500	1500	2100	8500	20000	40000	128000	250000
Network Latency (ns)	200000	100000	30000	10000	1300	1300	1000	500

Argo in a nutshell

- VIPS-DSM for distributed systems
  - User-space implementation
  - Runs Pthreads (DRF programs)
    - Small porting effort to fully exploit new synchronization system and optimize synchronization performance
  - Page-based DSM (uses virtual memory faults for misses)
  - Pages have a home node (limitation: naïve distribution)
  - MPI is the “network layer” (limitation: only need RDMA)

Components of Argo

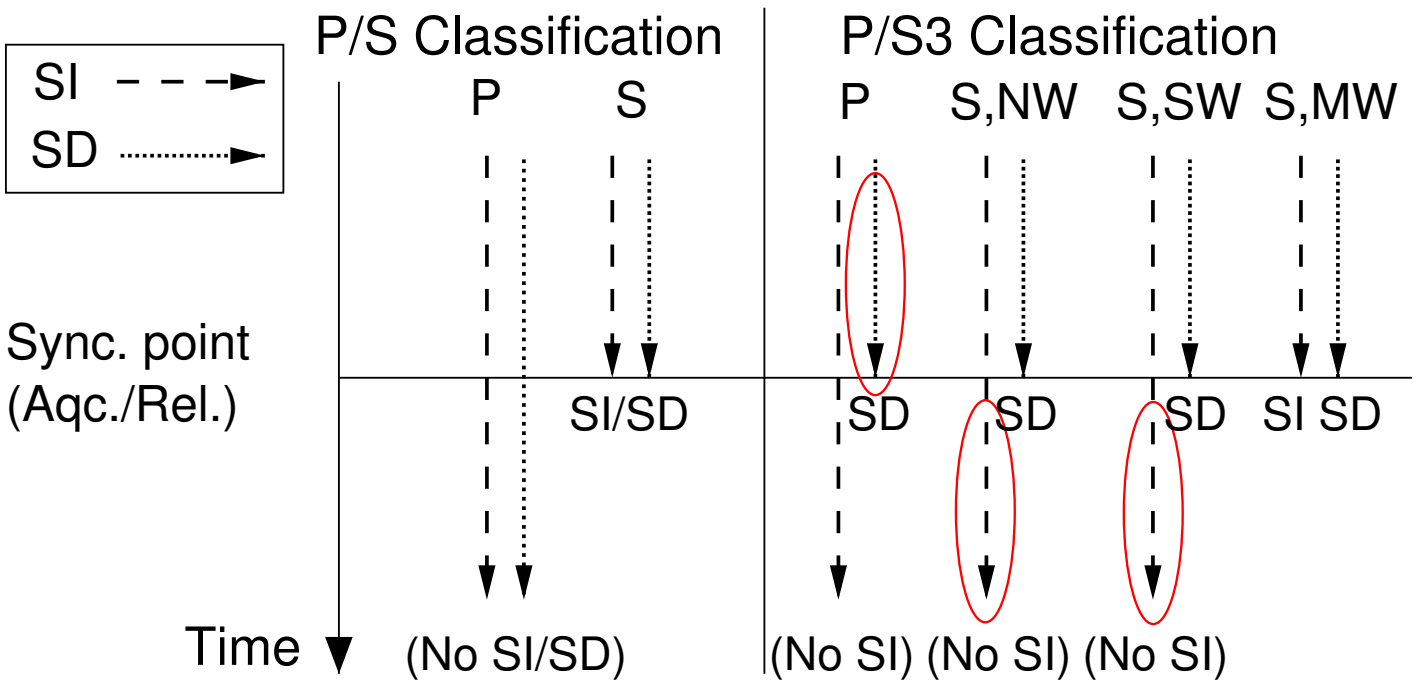
- Carina: VIPS-DSM coherence
- Pyxis: Classification directories
- Vela: Hierarchical Queue Delegation Locking system

CARINA & PYXIS: COHERENCE & DIRECTORIES

- Modified VIPS: SI & SD
  - Strictly request response for DRF accesses
- Pyxis classification directories cached at nodes
  - NO message handlers to classify pages and propagate classification changes
  - Requestors are responsible to update classification at remote nodes (P$\rightarrow$S, requestor updates private owner)

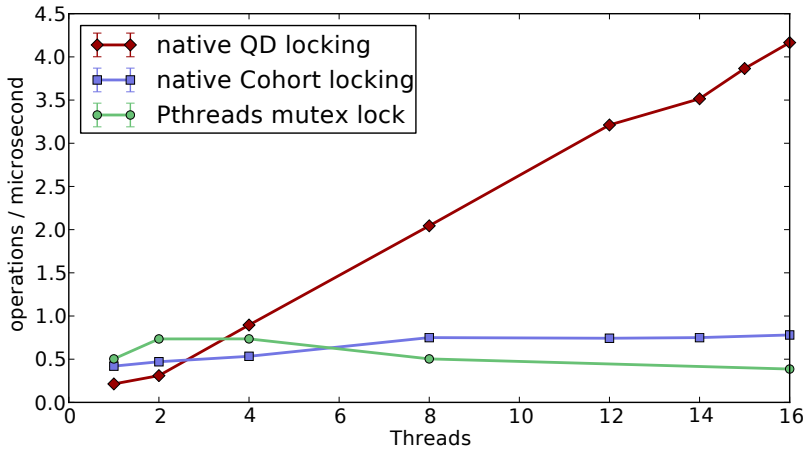
CARINA & PYXIS: COHERENCE & DIRECTORIES

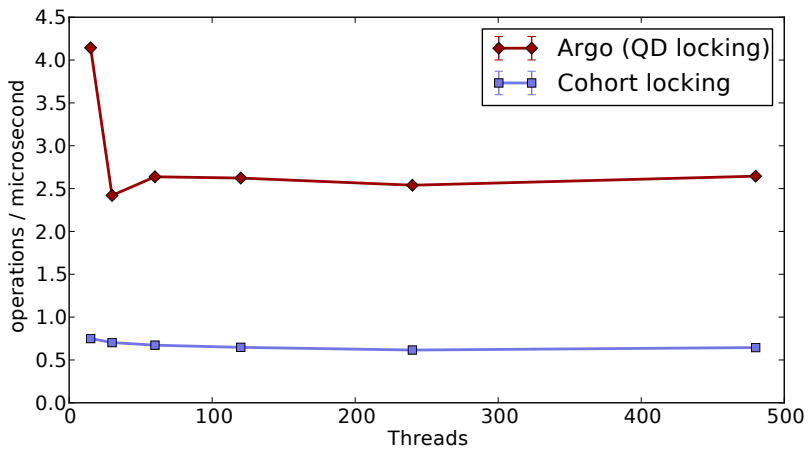
- Classification:
  - Only for Global shared memory (Gmalloc'ed)
  - Adds classification for writers
  - Private, Shared-NW (No Writers), Shared-SW (Single Writer), Shared-MW (Multiple Writers)



VELA: ARGO'S SYNCHRONIZATION SYSTEM

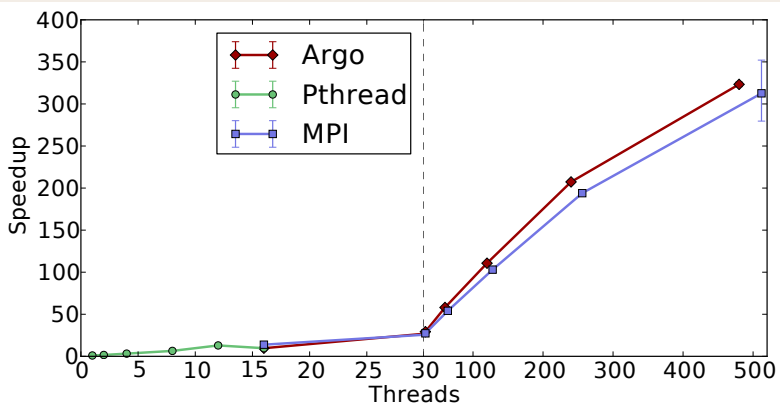
- The trouble with distributed critial section (CS) execution: Serialized execution that migrates from node to node!
  - Forces data accessed in CS to migrate too
  - Must SI on Lock, SD on Unlock
- Solution: Queue Delegation Locking [SPAA'14, EuroPar'14]
  - Delegate the execution of the CS to the current holder of the lock (up to a point)
- Hierarchical QDL: Delegate only locally



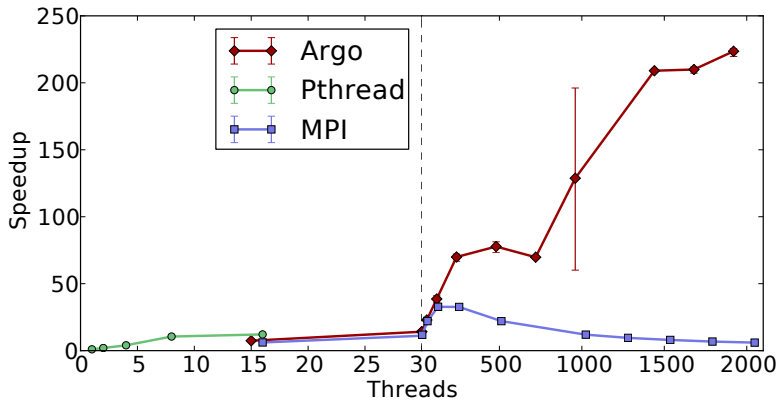


RESULTS

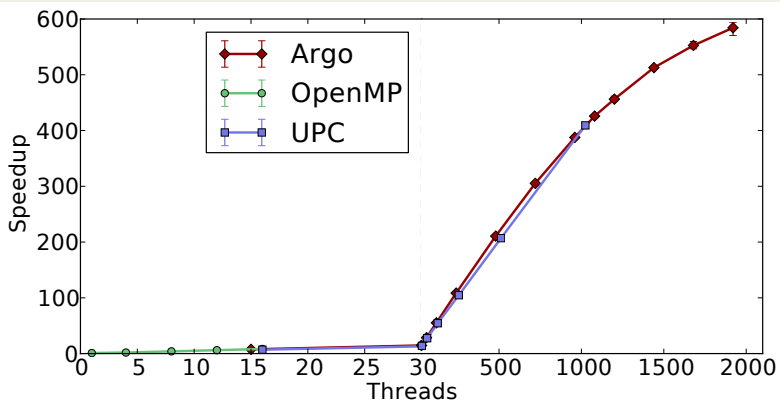
NBODY



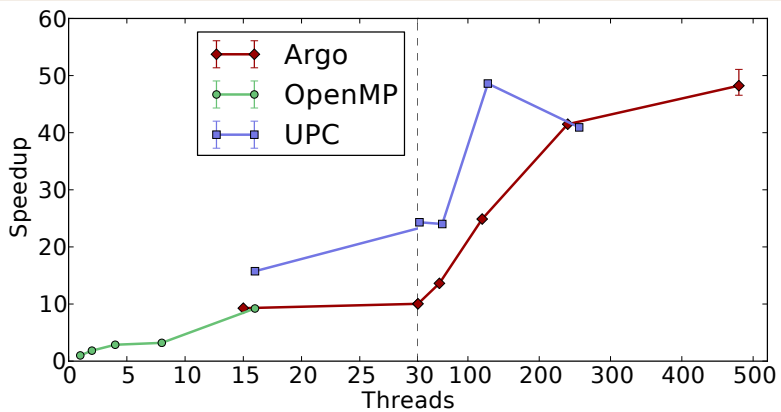
BLACKSCHOLES



EP CLASS D



CG CLASS C



Recap

- In this talk:
  - VIPS-M: Simple Request-Response Protocols [PACT'12]
  - VIPS-V: Virtual Cache Coherence [ISCA'13]
  - VIPS-H: Clustered Hierarchies [HPCA'15]
  - Callbacks: Efficient Spin-Waiting [ISCA'15]
  - Argo: Distributed Shared Memory [HPDC'15]
- Other VIPS works:
  - VIPS-B: Bus coherence [SoCC'12]
  - Fast&Furious: Data-Race Detector [PARMA-DITAM'15]
  - VIPS-GC: Generational Coherence [TACO'15]
  - Dir_1-SISD: Self-Contained Directories [PACT'15]
  - VIPS-G: CPU-GPU Coherence [TACO'16]

PROTOCOLOS DE COHERENCIA SENSIBLES AL MODELO DE CONSISTENCIA

Alberto Ros Manuel E. Acacio

Universidad de Murcia
{aros,meacacio}@ditec.um.es

13 de febrero de 2015

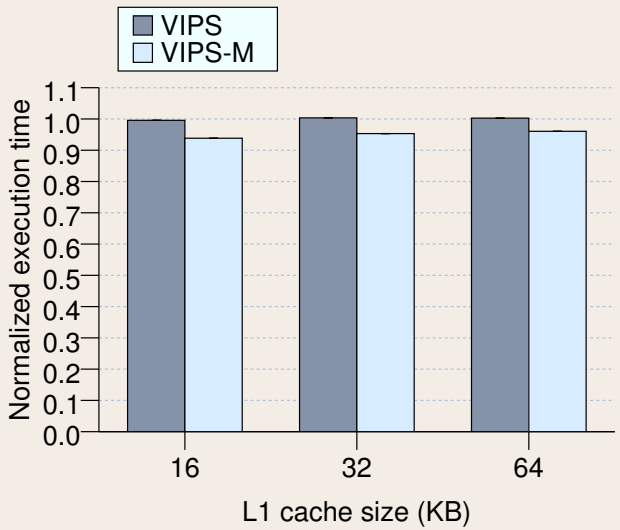
SIMULATION ENVIRONMENT

- SIMICS (functional simulation) + GEMS (memory timing) + GARNET (network)
- CACTI 6.5 for 32nm technology
- Simulated a 16-tile multicore
  - 32KB 4-way I&D L1s, 8MB (512KB/bank) 16-way L2 (LLC)
  - 16-entry MSHRs with 1000-cycle timeout
- SPLASH-2, scientific, and PARSEC benchmarks.

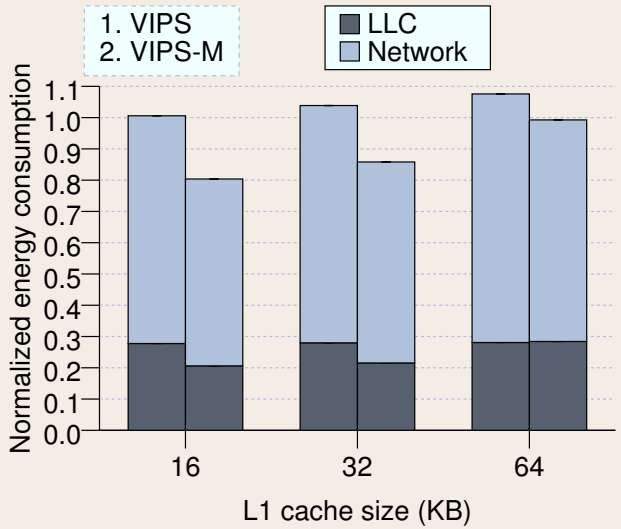
Protocol	Invalidations	Directory	Indirection	L1 base states
Hammer	Broadcast	None	Yes	5 (MOESI)
Directory	Multicast	Full-map	Yes	4 (MESI)
Write-Through	Multicast	Full-map	Only write misses	2 (VI)
VIPS	Multicast	Full-map	Only for write misses	2 (VI)
VIPS-M	None	None	No	2 (VI)

Evaluation

Performance 16KB–64KB L1



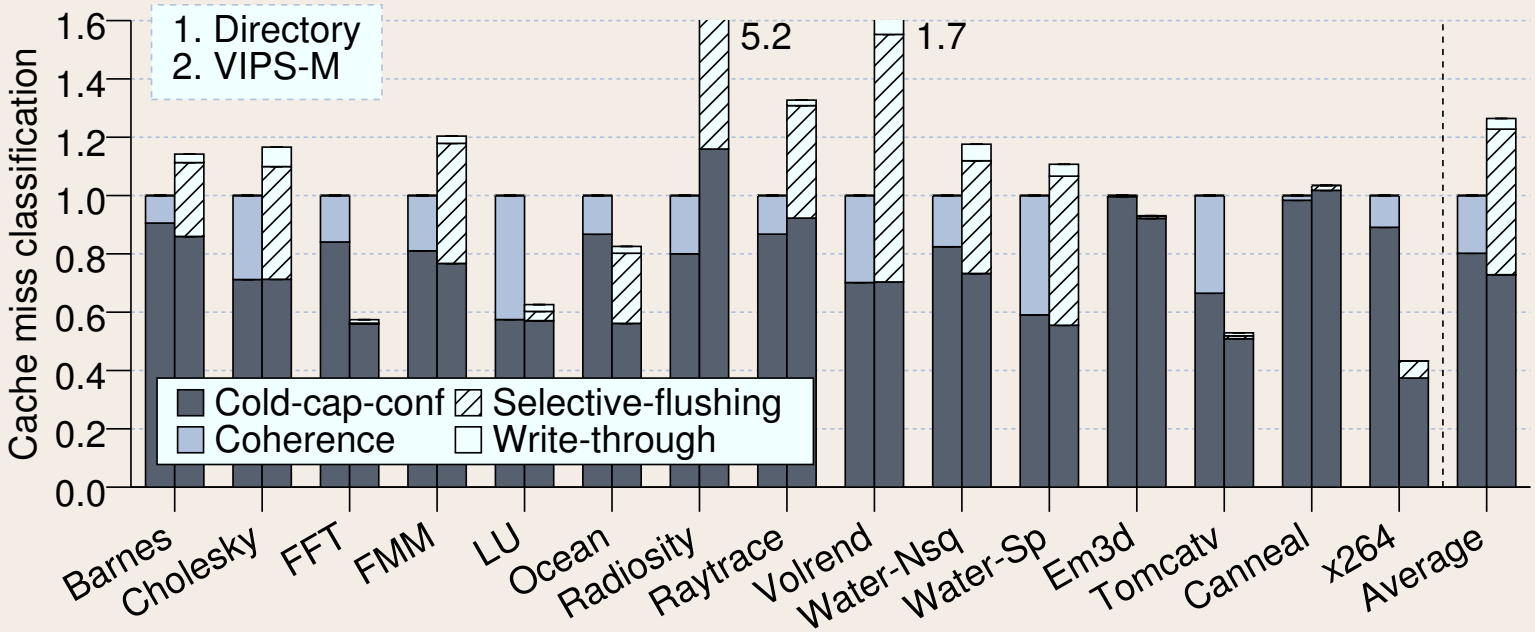
Energy 16KB–64KB L1



RESULTS

- Cold-cap-conf misses decrease due to the lack of write misses for DRF blocks
- Misses due to write throughs are not significant

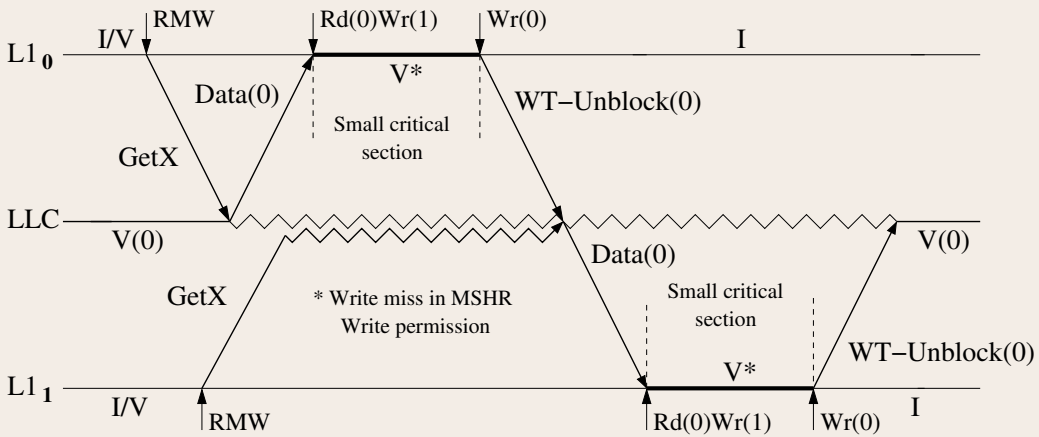
CACHE MISSES NORMALIZED W.R.T. DIRECTORY



VIPS-M

- Works very well for small critical sections

ATOMIC RMW TRANSACTIONS FOR SHARED BLOCKS



- Exponential back-off required for power reasons for large critical sections
- Considering hardware synchronization all protocols will be reduced to request-response transactions