

# PROTOCOLOS DE COHERENCIA SENSIBLES AL MODELO DE CONSISTENCIA

**Alberto Ros**   Manuel E. Acacio

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

19 de febrero de 2016

## 1 CONSISTENCIA DE MEMORIA

- Introducción
- Sequential Consistency
- Total Store Order
- Weak Consistency
- Release Consistency

## 2 PROTOCOLOS SENSIBLES A LA CONSISTENCIA

- Motivación
- Propuestas

## 3 LA FAMILIA DE PROTOCOLOS VIPS

- VIPS: Dynamic write policy
- VIPS-M: Self-invalidation
- VIPS-V: Virtual caches
- VIPS-H: Hierarchical systems
- Callbacks: Efficient spin-loops
- Argo: Distributed shared memory

# OUTLINE

## 1 CONSISTENCIA DE MEMORIA

- Introducción
- Sequential Consistency
- Total Store Order
- Weak Consistency
- Release Consistency

## 2 PROTOCOLOS SENSIBLES A LA CONSISTENCIA

- Motivación
- Propuestas

## 3 LA FAMILIA DE PROTOCOLOS VIPS

- VIPS: Dynamic write policy
- VIPS-M: Self-invalidation
- VIPS-V: Virtual caches
- VIPS-H: Hierarchical systems
- Callbacks: Efficient spin-loops
- Argo: Distributed shared memory

# ALGORITMO DE DEKKER

## ALGORITMO DE DEKKER

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

```
X = 1;
```

```
$r0 = Y;
```

```
Y = 1;
```

```
$r1 = X;
```

- ¿Puede dar el resultado  $X=0$ ,  $Y=0$ ?

# ALGORITMO DE DEKKER

## ALGORITMO DE DEKKER

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

```
X = 1;
```

```
$r0 = Y;
```

```
Y = 1;
```

```
$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<sup>1</sup>

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.

<sup>1</sup> 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<sup>1</sup>

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;  
$r0 = Y;
```

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

<sup>1</sup> L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs", IEEE Transactions on Computers, Sept. 1979.

# SEQUENTIAL CONSISTENCY (SC)

- Formalmente, siendo  $\langle p$  el orden en el programa y  $\langle m$  el orden en memoria:
  - Si  $L(a) \langle p L(b) \Rightarrow L(a) \langle m L(b)$  /\* Load $\rightarrow$ Load \*/
  - Si  $L(a) \langle p S(b) \Rightarrow L(a) \langle m S(b)$  /\* Load $\rightarrow$ Store \*/
  - Si  $S(a) \langle p S(b) \Rightarrow S(a) \langle m S(b)$  /\* Store $\rightarrow$ Store \*/
  - Si  $S(a) \langle p L(b) \Rightarrow S(a) \langle m L(b)$  /\* Store $\rightarrow$ Load \*/
  - Igual para las operaciones atómicas (RMW)
  
- Modelo empleado en el MIPS R10000

# SEQUENTIAL CONSISTENCY (SC)

- Formalmente, siendo  $\langle p$  el orden en el programa y  $\langle m$  el orden en memoria:
  - Si  $L(a) \langle p L(b) \Rightarrow L(a) \langle m L(b)$  /\* Load→Load \*/
  - Si  $L(a) \langle p S(b) \Rightarrow L(a) \langle m S(b)$  /\* Load→Store \*/
  - Si  $S(a) \langle p S(b) \Rightarrow S(a) \langle m S(b)$  /\* Store→Store \*/
  - Si  $S(a) \langle p L(b) \Rightarrow S(a) \langle m L(b)$  /\* Store→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;  
$r0 = Y;
```

```
Y = 1;  
$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;  
$r0 = Y;
```

```
Y = 1;  
$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→Load \*/
- Si  $L(a) <_p S(b) \Rightarrow L(a) <_m S(b)$  /\* Load→Store \*/
- Si  $S(a) <_p S(b) \Rightarrow S(a) <_m S(b)$  /\* Store→Store \*/
- /\* No Store→Load! \*/
- Orden en las operaciones atómicas (RMW)

# TOTAL STORE ORDER (TSO)

- Formalmente:

- Si  $L(a) <_p L(b) \Rightarrow L(a) <_m L(b)$  /\* Load→Load \*/
- Si  $L(a) <_p S(b) \Rightarrow L(a) <_m S(b)$  /\* Load→Store \*/
- Si  $S(a) <_p S(b) \Rightarrow S(a) <_m S(b)$  /\* 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>B</b>	X	X
	RMW	X	X	X

- **B** quiere decir que en caso de que una lectura encuentre 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	<b>X</b>
	Store	<b>B</b>	X	X	<b>X</b>
	RMW	X	X	X	<b>X</b>
	Fence	<b>X</b>	<b>X</b>	<b>X</b>	<b>X</b>

# FENCES

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

## CÓDIGO QUE SE COMPORTA COMO SC BAJO TSO

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

```
Y = 1;  
Fence;  
$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;  
flag = 1;
```

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

MOTIVACIÓN WEAK CONSISTENCY (WC)<sup>2</sup>

## SINCRONIZACIÓN CON FLAGS

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

```
X = 1;  
Y = 2;  
flag = 1;
```

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

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

<sup>2</sup> M. Dubois *et al.*, "Memory Access Buffering in Multiprocessors", ISCA, June 1986.

MOTIVACIÓN WEAK CONSISTENCY (WC)<sup>2</sup>

## SINCRONIZACIÓN CON FLAGS

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

```
X = 1;  
Y = 2;  
flag = 1;
```

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

<sup>2</sup> M. Dubois *et al.*, "Memory Access Buffering in Multiprocessors", ISCA, June 1986.

# WEAK CONSISTENCY (WC)

- La sincronización actua como fences
- Sequential Consistency for Data-Race-Free programs (SC for DRF)<sup>3</sup>
- La sincronización debe ser expuesta al hardware

<sup>3</sup> S. V. Adve and M. D. Hill. “Weak Ordering—A New Definition”, ISCA, May 1990

# WEAK CONSISTENCY (WC)

- La sincronización actua como fences
- Sequential Consistency for Data-Race-Free programs (SC for DRF)<sup>3</sup>
- La sincronización debe ser expuesta al hardware

## REGLAS DE ORDENACIÓN WC

		Op2			
		Load	Store	RMW	Sync
Op1	Load	<b>A</b>	<b>A</b>	<b>A</b>	X
	Store	B	<b>A</b>	<b>A</b>	X
	RMW	<b>A</b>	<b>A</b>	<b>A</b>	X
	Sync	X	X	X	X

<sup>3</sup> S. V. Adve and M. D. Hill. "Weak Ordering—A New Definition", ISCA, May 1990

# MOTIVACIÓN RELEASE CONSISTENCY (RC)<sup>4</sup>

## SINCRONIZACIÓN CON FLAGS

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

```
X = 1;
```

```
Y = 2;
```

```
flag = 1;
```

```
while (flag == 0);
```

```
$r1 = X;
```

```
$r2 = Y;
```

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

<sup>4</sup> K. Gharachorloo *et al.*, "Memory Consistency and Event Ordering in Scalable Shared-Memory", ISCA, May 1990.

# MOTIVACIÓN RELEASE CONSISTENCY (RC)<sup>4</sup>

## 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

<sup>4</sup> 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	<b>ACQ</b>	<b>REL</b>
Op1	Load	A	A	A	<b>A</b>	X
	Store	B	A	A	<b>A</b>	X
	RMW	A	A	A	<b>A</b>	X
	<b>ACQ</b>	X	X	X	X	X
	<b>REL</b>	<b>A</b>	<b>A</b>	<b>A</b>	X	X

# OUTLINE

## 1 CONSISTENCIA DE MEMORIA

- Introducción
- Sequential Consistency
- Total Store Order
- Weak Consistency
- Release Consistency

## 2 PROTOCOLOS SENSIBLES A LA CONSISTENCIA

- Motivación
- Propuestas

## 3 LA FAMILIA DE PROTOCOLOS VIPS

- VIPS: Dynamic write policy
- VIPS-M: Self-invalidation
- VIPS-V: Virtual caches
- VIPS-H: Hierarchical systems
- Callbacks: Efficient spin-loops
- Argo: Distributed shared memory

# MOTIVACIÓN

- La consistencia de memoria normalmente usa el protocolo de coherencia de cache como una “caja negra”
- Esta caja no rompe el modelo de consistencia de la máquina
  - Garantiza SC para un sistema que no reordena accesos
- Esto se hace porque es más fácil 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-invalidación
  - 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.*<sup>5</sup> 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.*<sup>6</sup> 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

<sup>5</sup> R. L. Lee *et al.*, "Multiprocessor Cache Design Considerations", ISCA, June 1987.

<sup>6</sup> H. Cheong and A. V. Vaidenbaum, "A Cache Coherence 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

<sup>7</sup> 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

<sup>8</sup> B. Choi *et al.*, “DeNovo: Rethinking the memory hierarchy for disciplined parallelism”, PACT, Oct. 2011.

# AUTO-INVALIDACIÓN SELECTIVA CON BLOOM FILTERS<sup>9</sup>

- **Self-downgrade** 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

<sup>9</sup> 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<sup>10,11,12,13,14</sup>

- 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<sup>11</sup>
- Ideal para jerarquías de cache parcialmente compartidas<sup>12</sup>
- Sincronización eficiente<sup>13</sup>
- Ideal para distributed shared memory<sup>14</sup>

<sup>10</sup> A. Ros and S. Kaxiras, “Complexity-effective multicore coherence”, PACT, 2012.

<sup>11</sup> S. Kaxiras and A. Ros, “A new perspective for efficient virtual-cache coherence”, ISCA, 2013.

<sup>12</sup> A. Ros *et al.*, “Hierarchical Private/Shared Classification: the Key to Simple and Efficient Coherence for Clustered Cache Hierarchies”, HPCA, 2015.

<sup>13</sup> A. Ros and S. Kaxiras, “Callback: Efficient Synchronization without Invalidation with a Directory Just for Spin-Waiting”, ISCA, 2015.

<sup>14</sup> 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<sup>15</sup> Y RC3<sup>16</sup>

- 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

<sup>15</sup> M. Elver and V. Nagarajan, "TSO-CC: Consistency directed cache coherence for TSO" HPCA, 2014.

<sup>16</sup> M. Elver and V. Nagarajan, "RC3: Consistency directed cache coherence for x86-64 with RC extensions" PACT, 2015.

SPEL<sup>17,18</sup>

- Protocolo de consistencia 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

<sup>17</sup> A. Ros and A. Jimborean, "A Dual-Consistency Cache Coherence Protocol", IPDPS, May 2015.

<sup>18</sup> A. Ros and A. Jimborean, "A Hybrid Static-Dynamic Classification for Dual-Consistency Cache Coherence", TPDS, Mar. 2016.

# OUTLINE

## 1 CONSISTENCIA DE MEMORIA

- Introducción
- Sequential Consistency
- Total Store Order
- Weak Consistency
- Release Consistency

## 2 PROTOCOLOS SENSIBLES A LA CONSISTENCIA

- Motivación
- Propuestas

## 3 LA FAMILIA DE PROTOCOLOS VIPS

- VIPS: Dynamic write policy
- VIPS-M: Self-invalidation
- VIPS-V: Virtual caches
- VIPS-H: Hierarchical systems
- Callbacks: Efficient spin-loops
- Argo: Distributed shared memory

# 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?

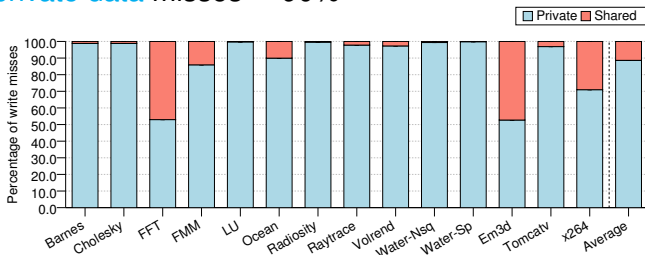


# MOTIVATION

- **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  $\approx 90\%$



# 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 → no extra misses
- Write-through for **S**hared blocks
  - Simple (only two states, **VI**)
  - Efficient → coherence misses

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

# PRIVATE/SHARED CLASSIFICATION

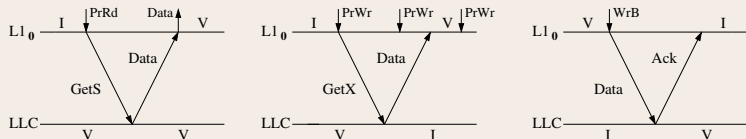
- Classify data (cache blocks) into **private** and **shared**
  - A-priori: Before issuing the coherence transaction we know if it is for a private or for a shared block
    - i.e., OS/TLB, compiler, application
    - Private and shared protocols completely independent
    - Can be verified separately
- 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

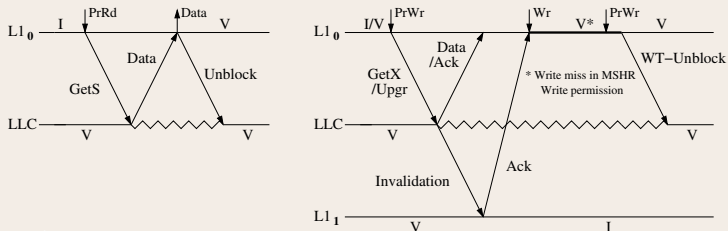
- We still have some (10%) write misses for shared blocks
- The obvious optimization to a WT policy is to delay WTs to coalesce as many writes as possible on the same cache line for performance
- Implementation on MSHRs
  - We add a hidden transient state (no races)
  - WTs triggered by timers (after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

# TWO INDEPENDENT PROTOCOLS

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



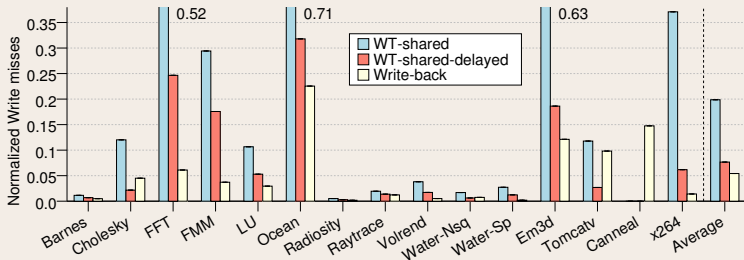
## READ AND WRITE TRANSACTIONS FOR SHARED BLOCKS



# 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

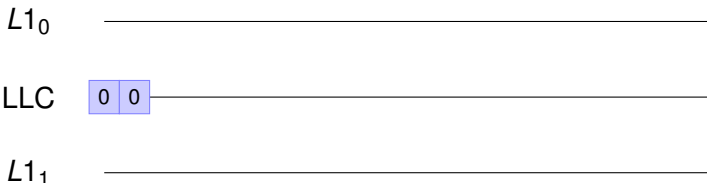
- Self-Invalidation of **shared** data from L1s
  - Selective Flush (SF) upon synchronization points
  - Self-Invalidation can be optimized to exclude read-only data and can be enhanced with speculation [*Speculative Cache Lookup/Coherence Decoupling, Huh et al., ASPLOS'04*]
- We eliminate **invalidations**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

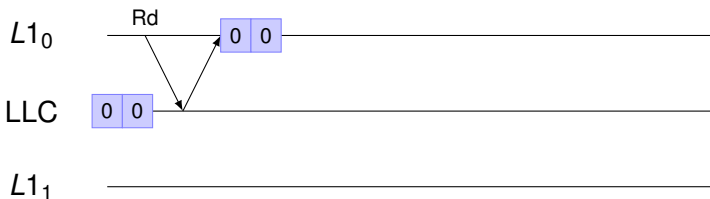
# MULTIPLE WRITERS & MERGE

- 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!



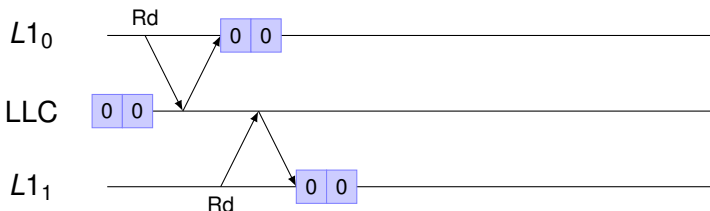
# MULTIPLE WRITERS & MERGE

- 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!



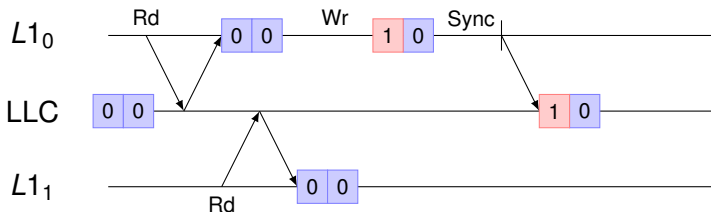
# MULTIPLE WRITERS & MERGE

- 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!



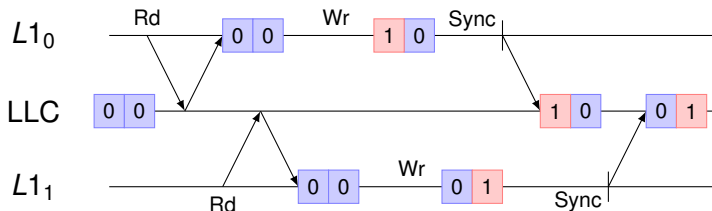
# MULTIPLE WRITERS & MERGE

- 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

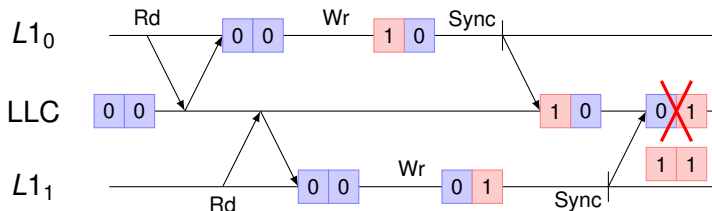
- Self-invalidation offers SC only for programs that are DRF at cache-line granularity
  - But not for DRF at word granularity
  - Does not work with false sharing
    - False sharing on a **cache line** while DRF at **word** granularity
    - We do not have invalidations on writes!





# 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!

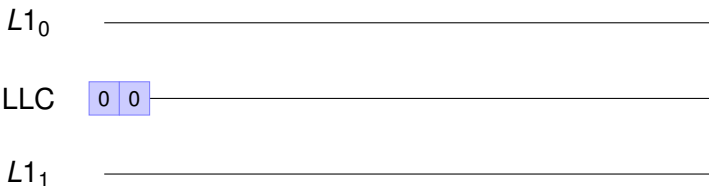


# MULTIPLE WRITERS & MERGE

- 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!

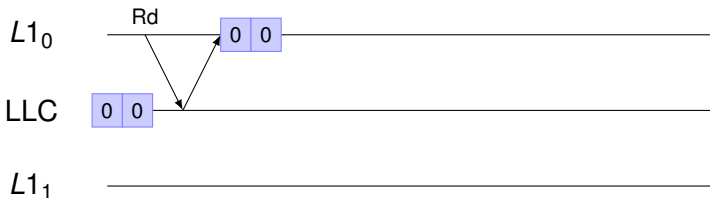
# MULTIPLE WRITERS & MERGE

- 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!



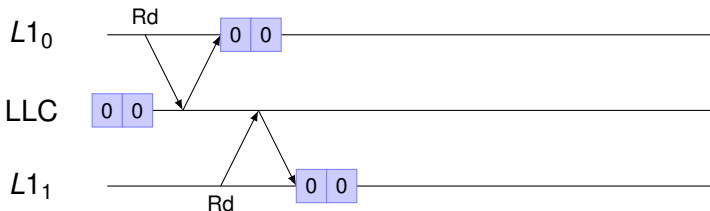
# MULTIPLE WRITERS & MERGE

- 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!



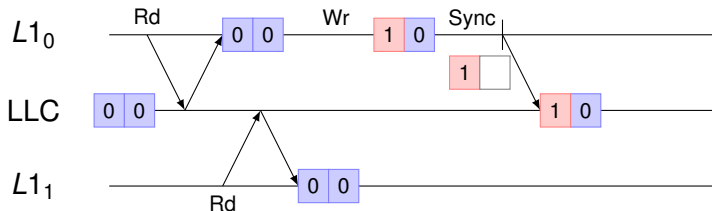
# MULTIPLE WRITERS & MERGE

- 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!



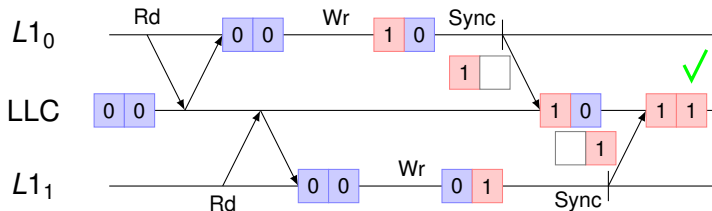
# MULTIPLE WRITERS & MERGE

- 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

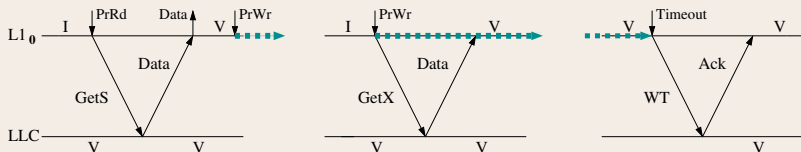
- This is solved by writing through only the modified words (send cache-line DIFFs) and merging them in the LLC
  - Cache structures are **not modified**
  - Written words bits only in MSHRs
    - Our “dirty” state is transient!



# 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

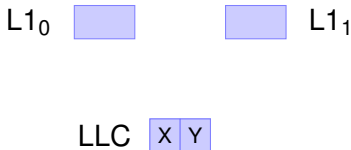
```

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

X = 1;
SIGNAL(cond);

```

\$r2 = Y;
WAIT(cond);
\$r1 = X;



- SIGNAL has release semantics
- WAIT has acquire semantics
- Acquire and release semantics guarantee that read returns the value generated by the write
  - SIGNAL ⇒ self-downgrade, WAIT ⇒ self-invalidate
- Implemented using atomic operations (not shown)

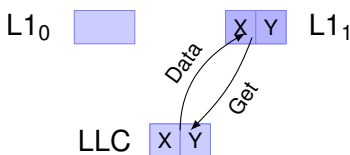
## EXAMPLE OF EXECUTION

## EXAMPLE OF DRF CODE

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

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

- SIGNAL has release semantics
- WAIT has acquire semantics
- Acquire 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)



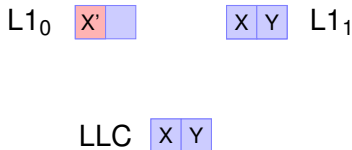
## EXAMPLE OF EXECUTION

## EXAMPLE OF DRF CODE

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

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

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



- SIGNAL has release semantics
- WAIT has acquire semantics
- Acquire 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)

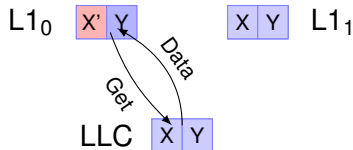
## EXAMPLE OF EXECUTION

## EXAMPLE OF DRF CODE

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

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

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



- SIGNAL has release semantics
- WAIT has acquire semantics
- Acquire and release semantics guarantee that read returns the value generated by the write
  - SIGNAL ⇒ self-downgrade, WAIT ⇒ self-invalidate
- Implemented using atomic operations (not shown)

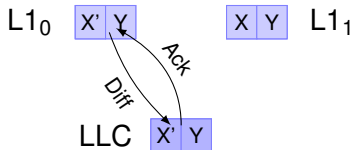
## EXAMPLE OF EXECUTION

## EXAMPLE OF DRF CODE

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

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

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



- SIGNAL has release semantics
- WAIT has acquire semantics
- Acquire and release semantics guarantee that read returns the value generated by the write
  - SIGNAL ⇒ self-downgrade, WAIT ⇒ self-invalidate
- Implemented using atomic operations (not shown)

# EXAMPLE OF EXECUTION

## EXAMPLE OF DRF CODE

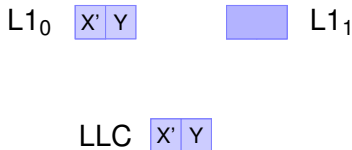
```

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

X = 1;
SIGNAL(cond);

```

	\$r2 = Y;
	WAIT(cond);
	\$r1 = X;



- SIGNAL has release semantics
- WAIT has acquire semantics
- Acquire 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)

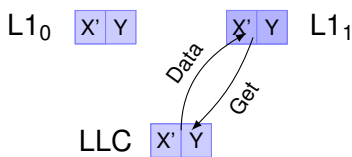
## EXAMPLE OF EXECUTION

## EXAMPLE OF DRF CODE

```

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

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



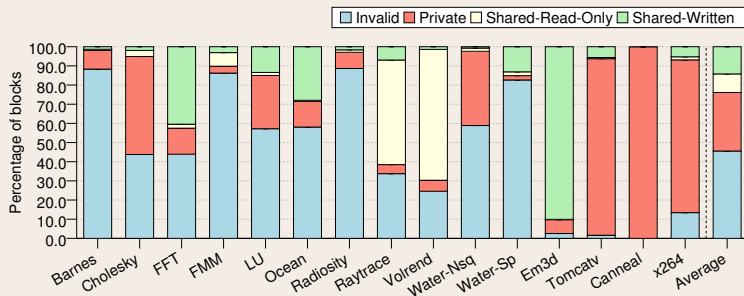
- SIGNAL has release semantics
- WAIT has acquire semantics
- Acquire and release semantics guarantee that read returns the value generated by the write
  - SIGNAL  $\Rightarrow$  self-downgrade, WAIT  $\Rightarrow$  self-invalidate
- Implemented using atomic operations (not shown)



# SELECTIVE FLUSHING

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

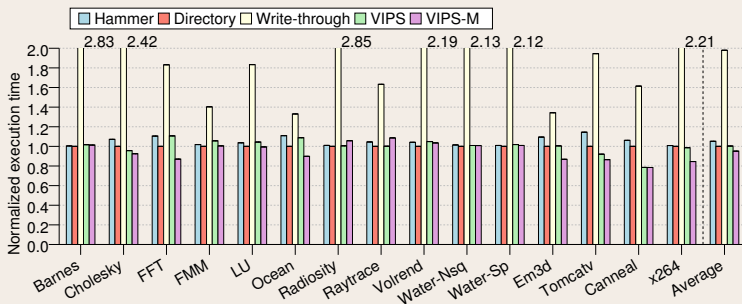
## LINES FOUND IN THE CACHE UPON A SELECTIVE FLUSHING



# 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

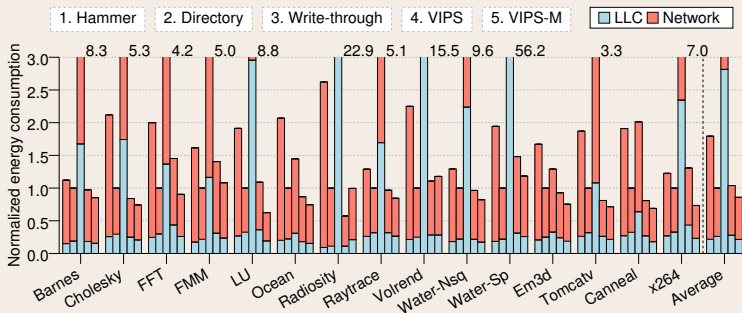
## NORMALIZED EXECUTION TIME W.R.T. DIRECTORY



# 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 & COHERENCE

- 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
    - ⇒ **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



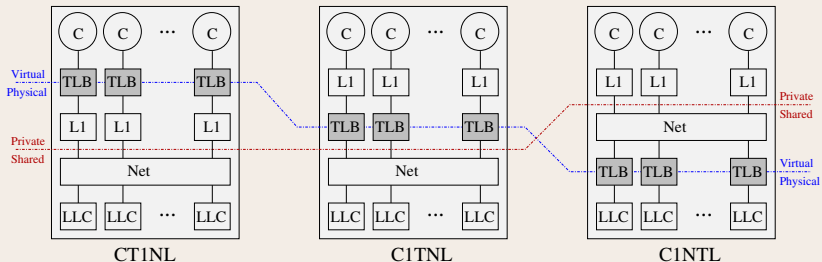
# SOLUTION FOR SYNONYMS

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 demapping/remapping
  - 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

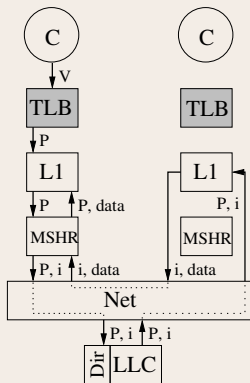
# DESIGN CHOICES FOR TLB PLACEMENT

- $cT_{1NL}$ : Physically-tagged L1 caches
- $c_{1T}NL$ : Virtual L1 caches, private TLBs
- $c_{1N}TL$ : Virtual L1 caches, shared TLBs



CT<sub>1</sub>NL: PHYSICALLY-TAGGED L1 CACHES

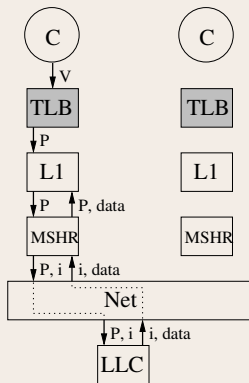
## MESI



No reverse translation

High TLB power consumption

## VIPS-M



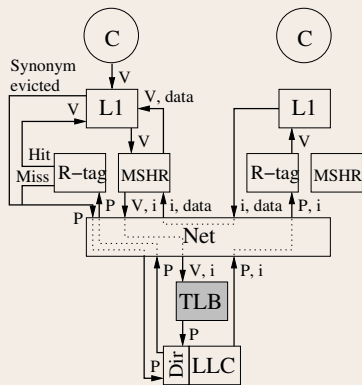
No reverse translation

High TLB power consumption

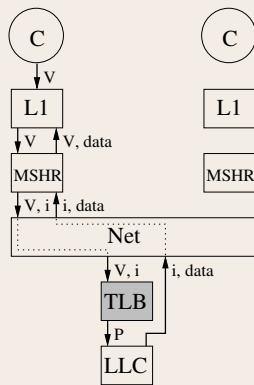


C<sub>1</sub>N<sub>1</sub>TL: VIRTUAL L1 CACHES, SHARED TLBS

## MESI



## VIPS-M

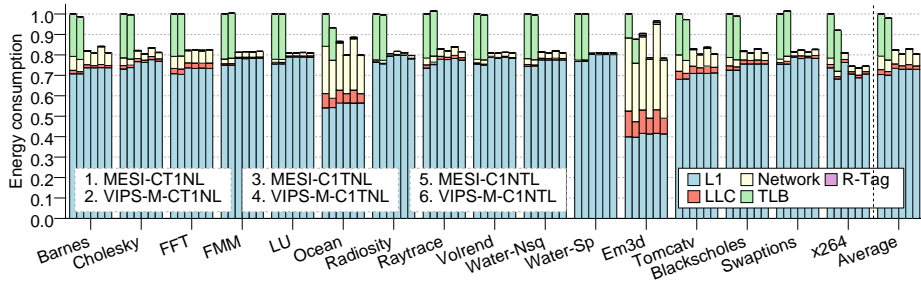


Low TLB power consumption  
 No replicated translations  
 R-trans, syn-check across Net

Low TLB power consumption  
 No replicated translations  
 No r-trans, no syn-check

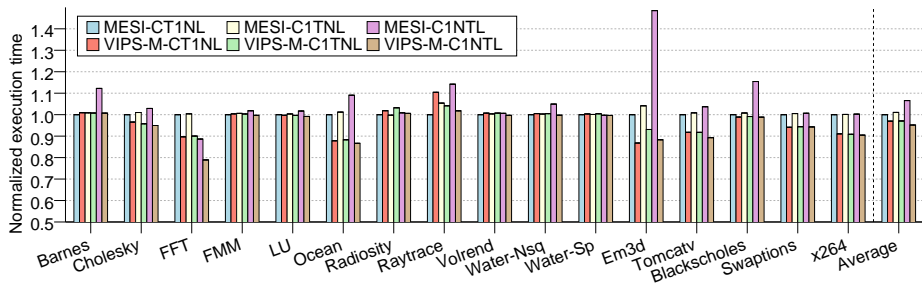
# ENRGY CONSUMPTION

- 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



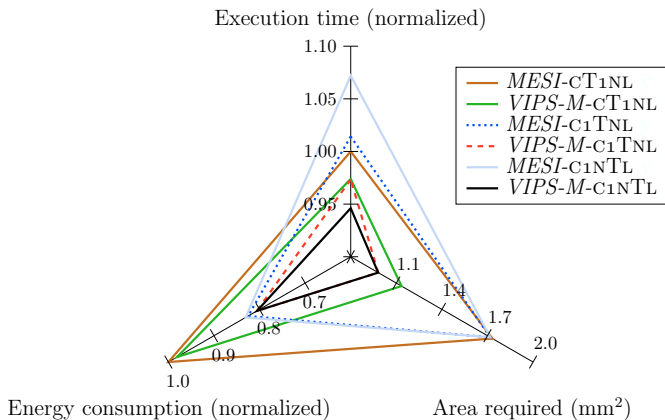
# EXECUTION TIME

- 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-C<sub>1</sub>N<sub>1</sub>TL is the best in terms of performance, energy and area

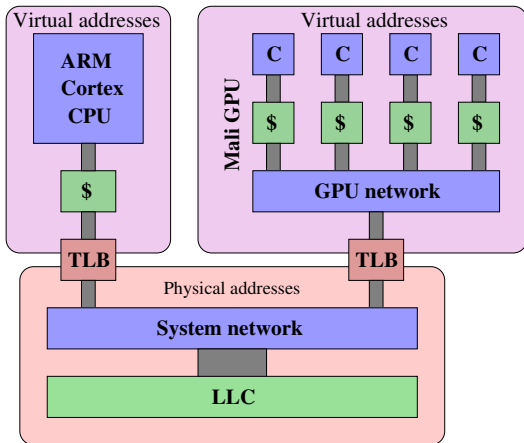




# VIPS-V CONCLUSIONS

- 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&amp;GPU COHERENCE

CPU: VIPS-M-C<sub>1</sub>TNLGPU: VIPS-M-C<sub>1</sub>N<sub>1</sub>TL

# MOTIVATION

- Clustered cache hierarchies are a natural strategy for reducing the **overhead** introduced by cache coherence protocols (e.g., storage and traffic)<sup>1</sup>

<sup>1</sup> 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)<sup>1</sup>
- 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)

<sup>1</sup> 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

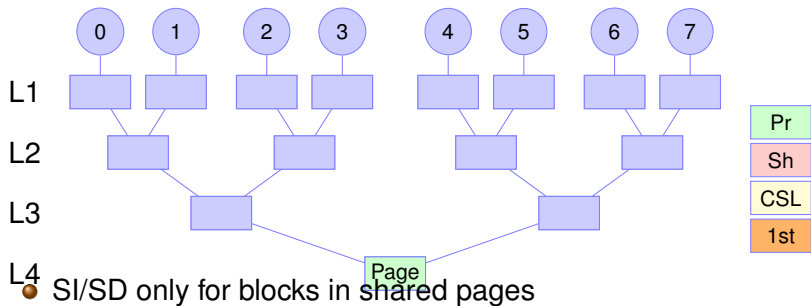
# CHALLENGE

- 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

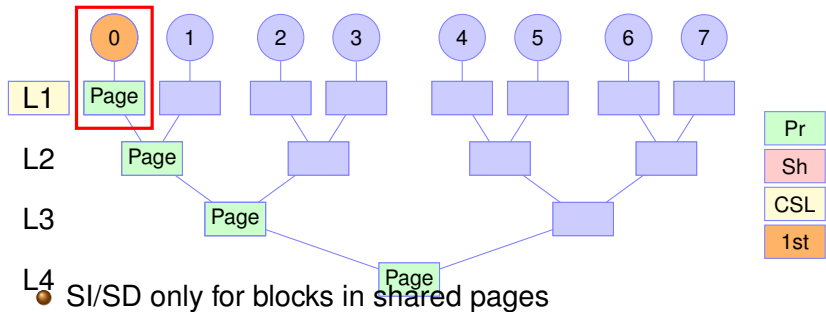
- We solve this problem by introducing the concept of **hierarchical P/S classification**
  - A block can be shared inside a cluster but be private outside
  - The level where this transition happens is the **common sharing level (CSL)**
  - Restrict SI/SD to shared blocks within a cluster
- **Result**
  - 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

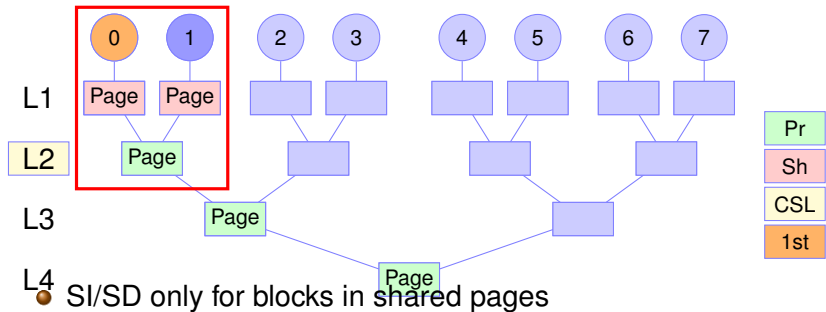




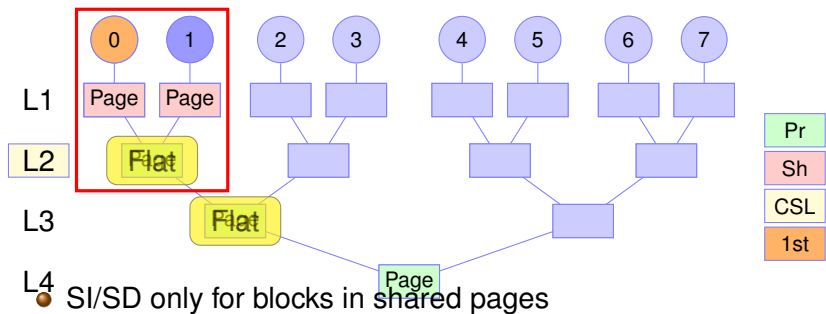
## COMMON SHARING LEVEL



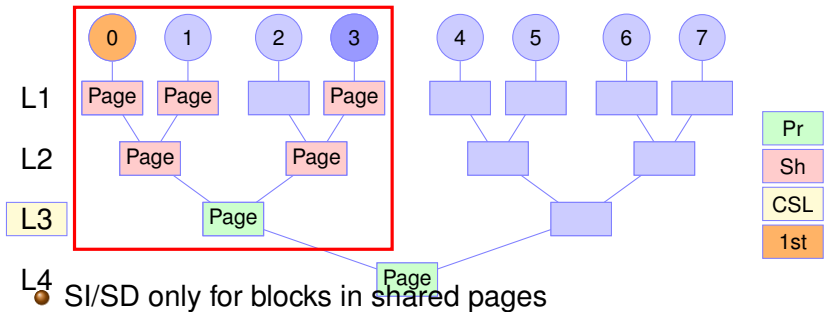
## COMMON SHARING LEVEL



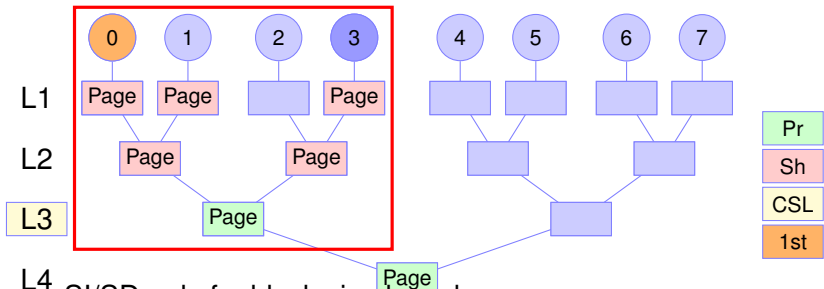
## COMMON SHARING LEVEL



## COMMON SHARING LEVEL



## COMMON SHARING LEVEL



- L4
- 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.		Frequency	Complexity	Efficiency
	(4×4)	(16×4)			
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 <sup>1</sup>	0.0006%	0.0005%	Very low	High	Low

<sup>1</sup> Performed upon changes in the CSL

# EXAMPLE

## EXAMPLE OF DRF CODE

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

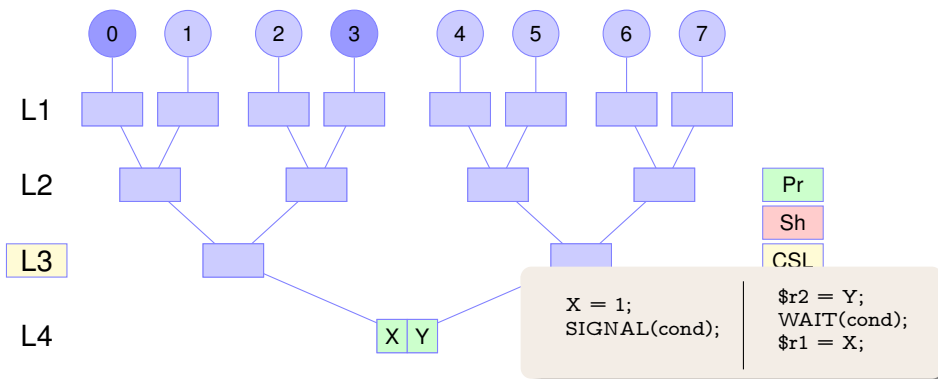
```
X = 1;  
SIGNAL(cond);
```

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

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

## VIPS-H: LOAD OPERATIONS

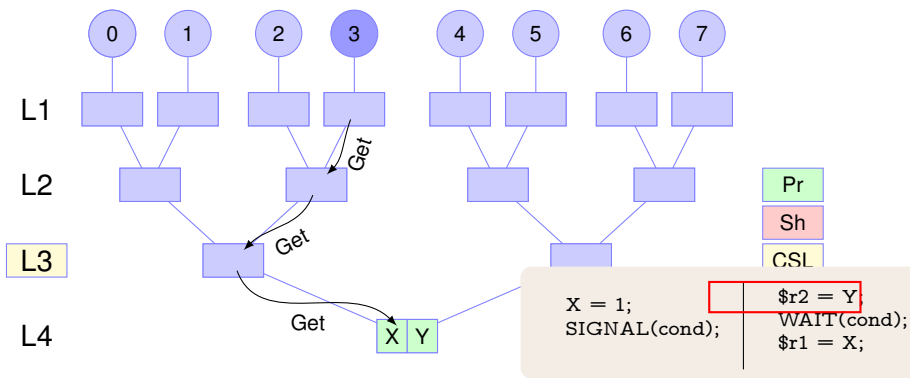
Event	% per instr.		Frequency	Complexity	Efficiency
	(4×4)	(16×4)			
Load instr	21.3251%	16.4038%	High	Low	High





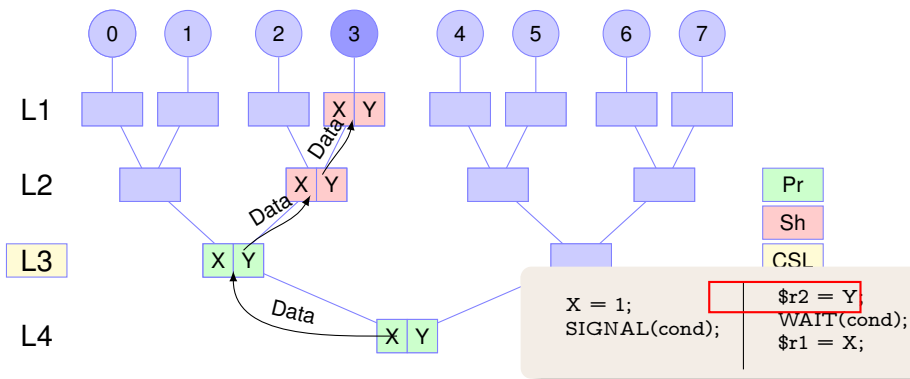
## VIPS-H: LOAD OPERATIONS

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



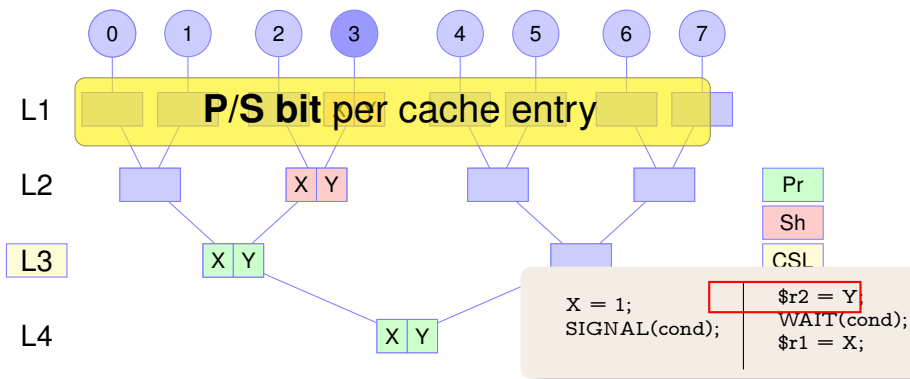
## VIPS-H: LOAD OPERATIONS

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



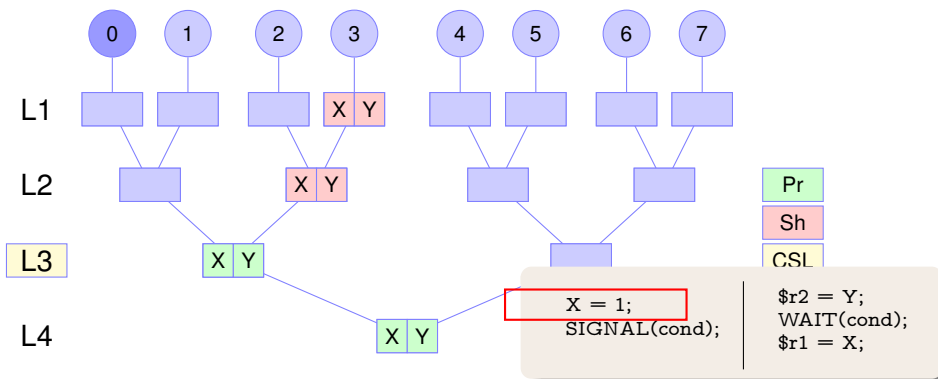
## VIPS-H: LOAD OPERATIONS

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



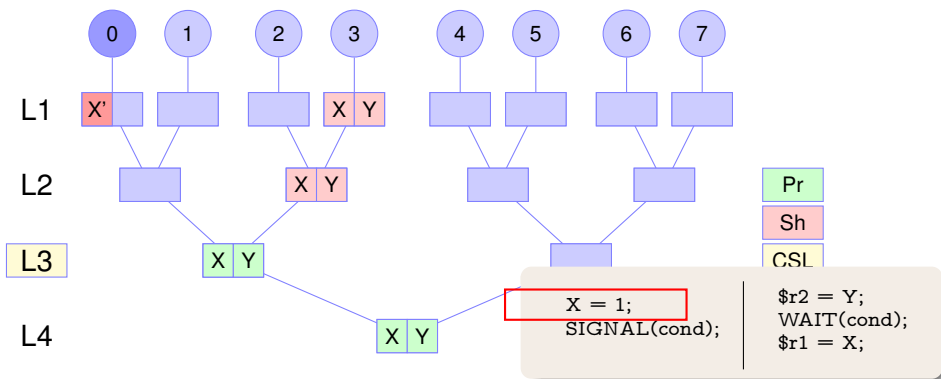
## VIPS-H: STORE OPERATIONS

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



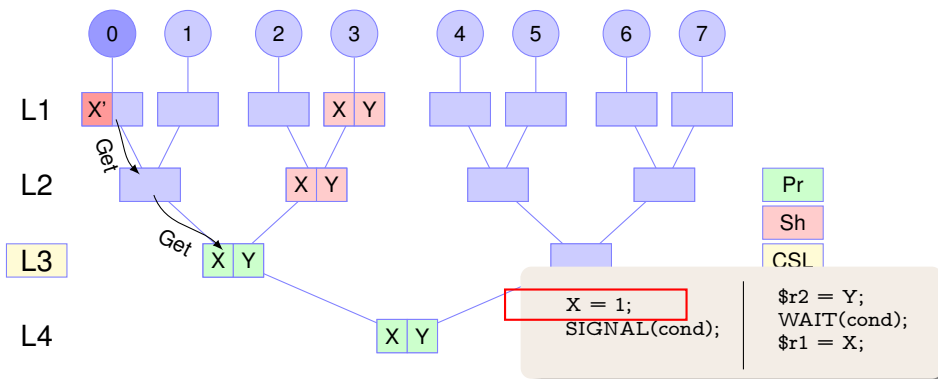
## VIPS-H: STORE OPERATIONS

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



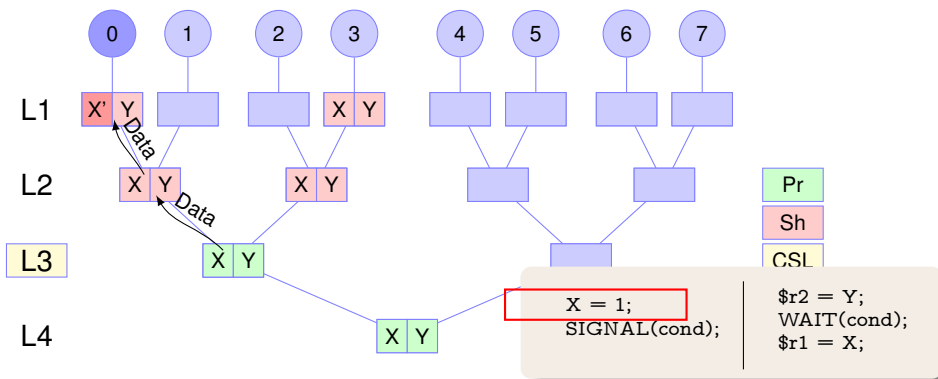
## VIPS-H: STORE OPERATIONS

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



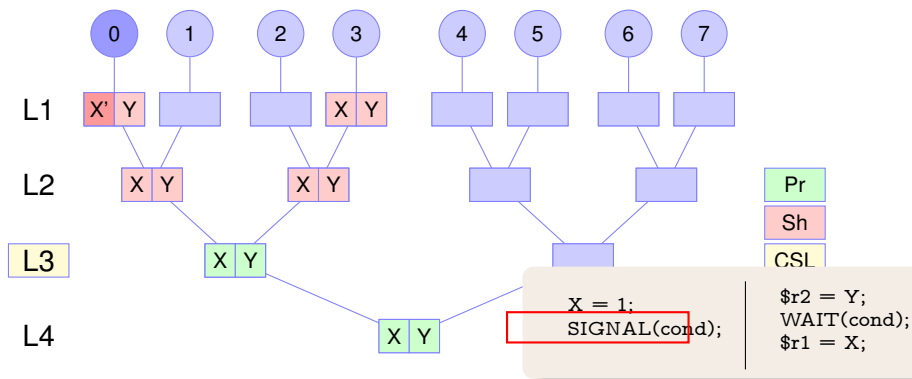
## VIPS-H: STORE OPERATIONS

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



## VIPS-H: SELF-DOWNGRADE

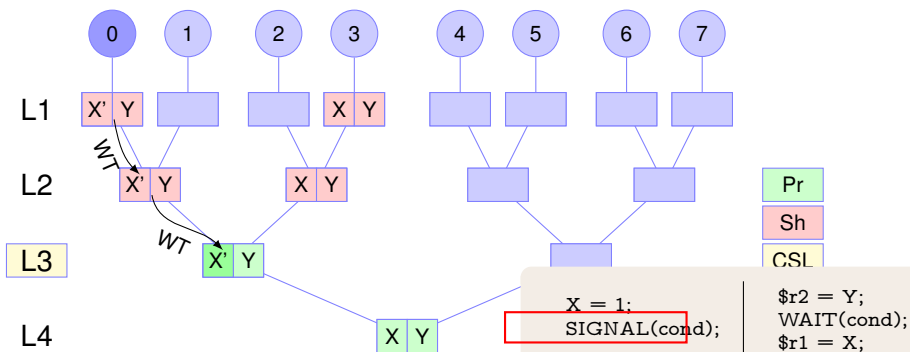
Event	% per instr.		Frequency	Complexity	Efficiency
	(4×4)	(16×4)			
SI&SD instr	0.0045%	0.0068%	Low	Medium	Low





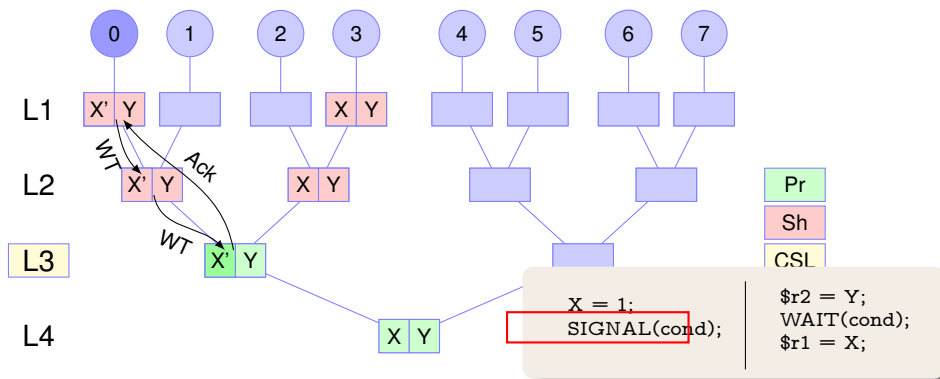
## VIPS-H: SELF-DOWNGRADE

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



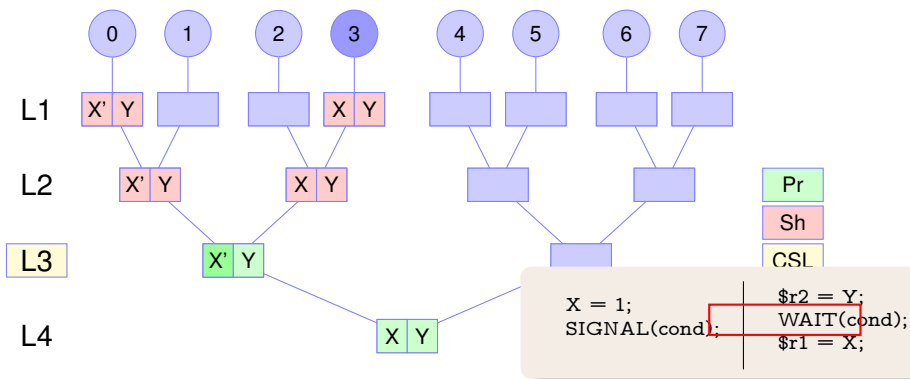
## VIPS-H: SELF-DOWNGRADE

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



## VIPS-H: SELF-INVALIDATION

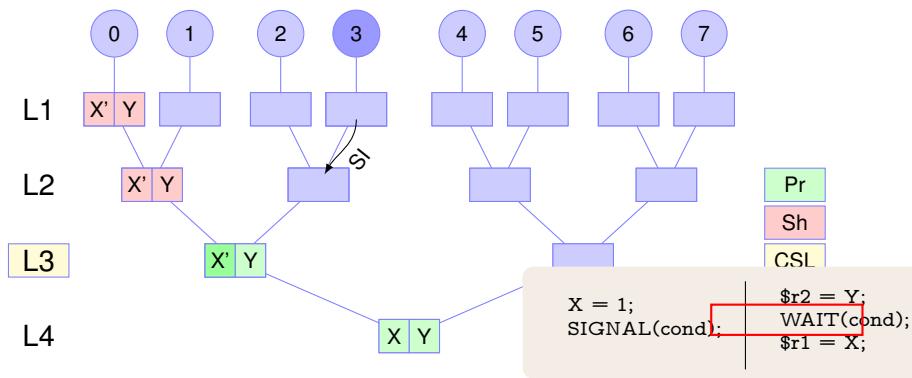
Event	% per instr.		Frequency	Complexity	Efficiency
	(4×4)	(16×4)			
SI&SD instr	0.0045%	0.0068%	Low	Medium	Low





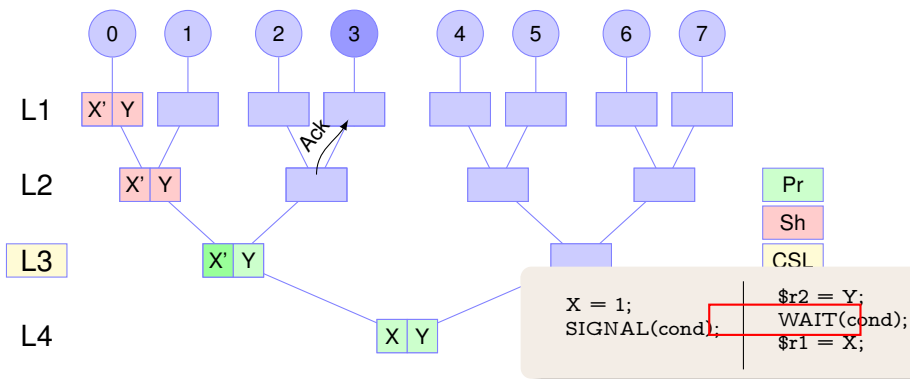
## VIPS-H: SELF-INVALIDATION

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



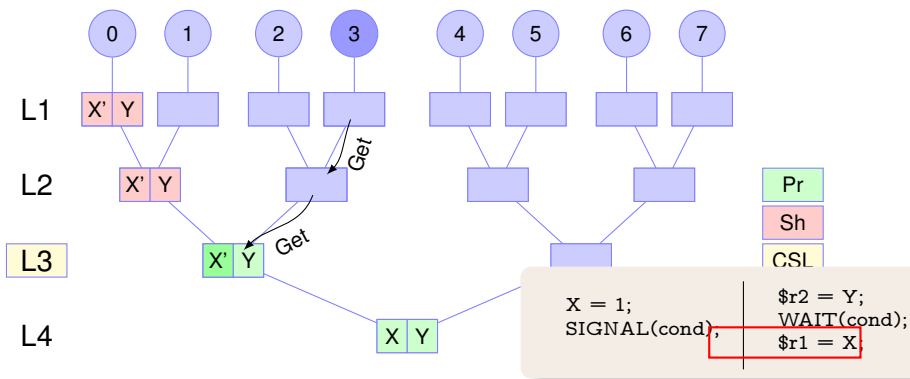
## VIPS-H: SELF-INVALIDATION

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



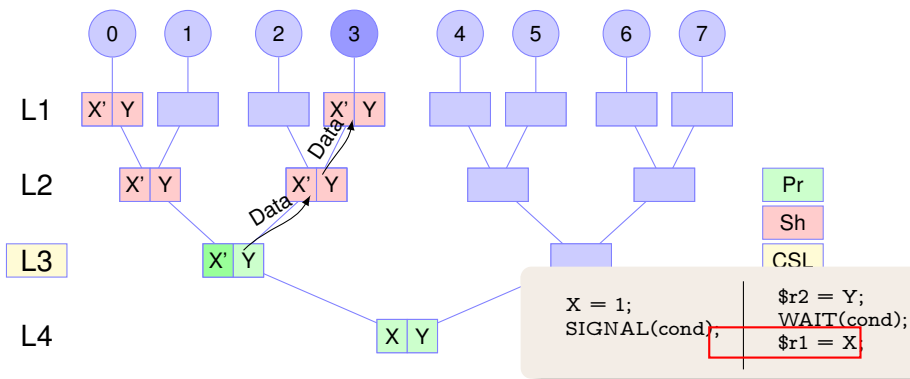
## VIPS-H: SELF-INVALIDATION

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



## VIPS-H: SELF-INVALIDATION

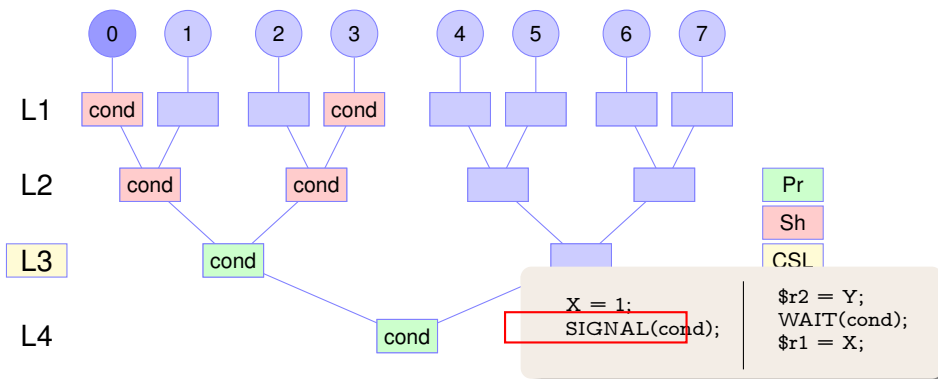
Event	% per instr.		Frequency	Complexity	Efficiency
	(4×4)	(16×4)			
SI&SD instr	0.0045%	0.0068%	Low	Medium	Low





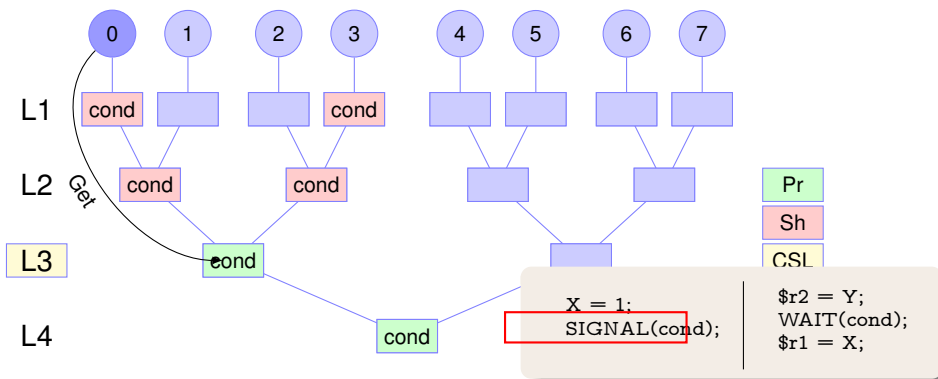
## VIPS-H: ATOMIC OPERATIONS

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



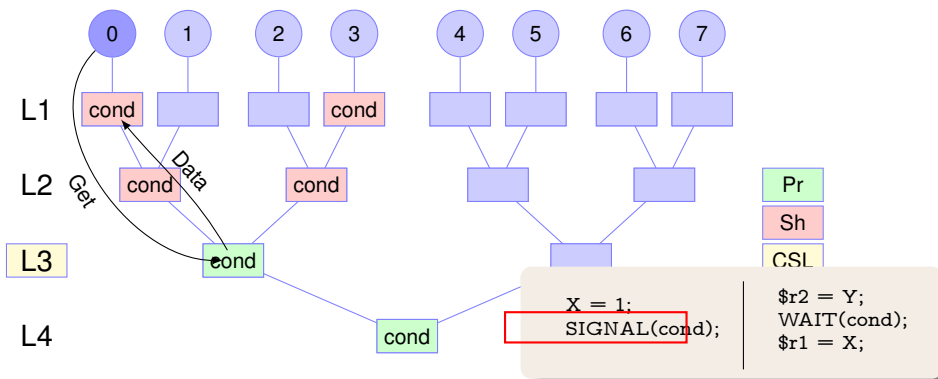
## VIPs-H: ATOMIC OPERATIONS

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



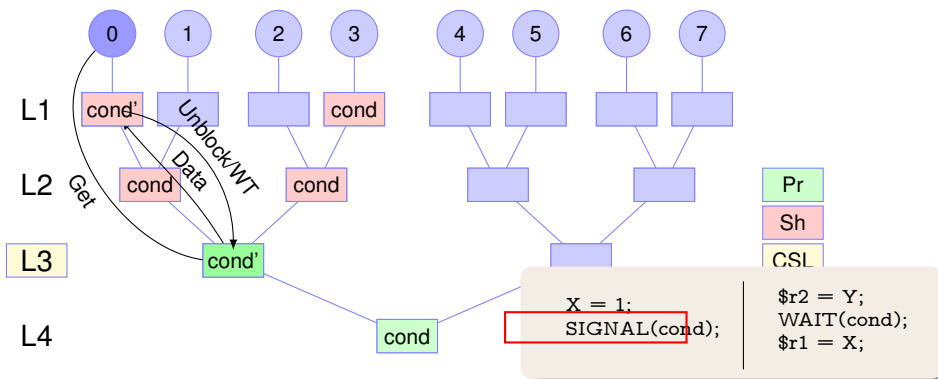
## VIPS-H: ATOMIC OPERATIONS

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



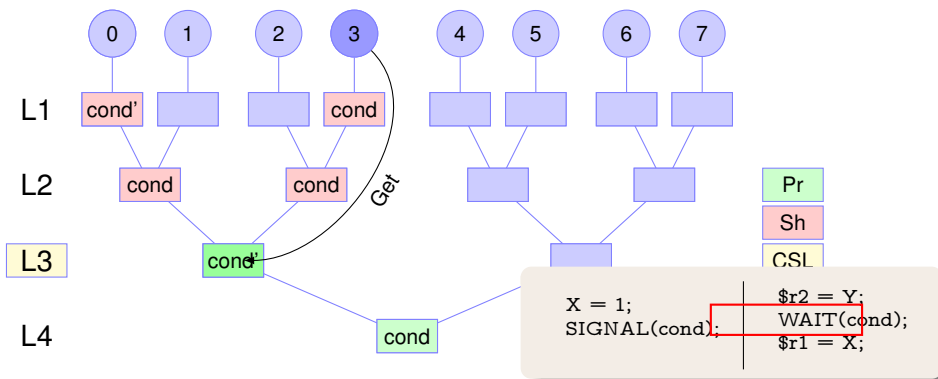
## VIPS-H: ATOMIC OPERATIONS

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



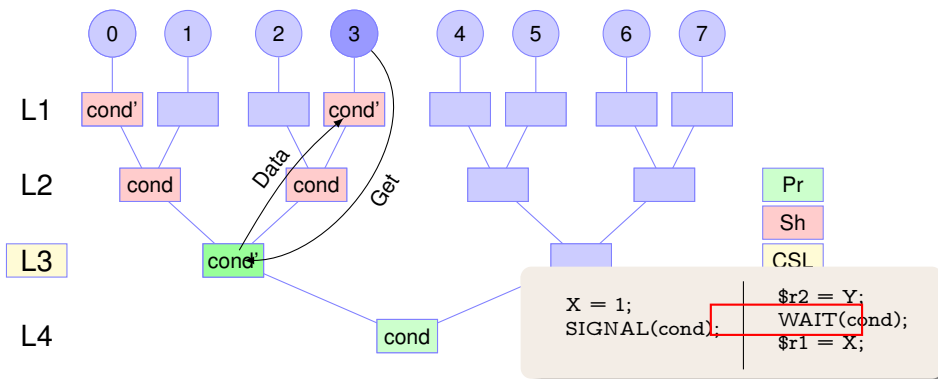
## VIPS-H: ATOMIC OPERATIONS

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



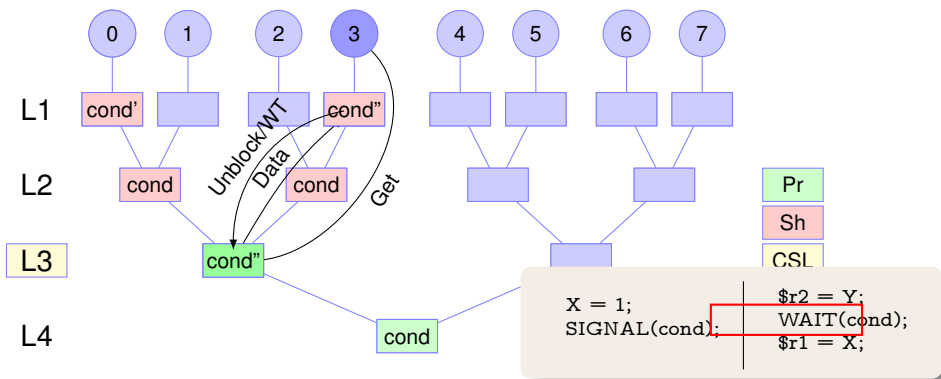
## VIP-S-H: ATOMIC OPERATIONS

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



## VIPS-H: ATOMIC OPERATIONS

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



# RECOVERIES: UPDATING CSL AND P/S

Event	% per instr.		Frequency	Complexity	Efficiency
	(4×4)	(16×4)			
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)

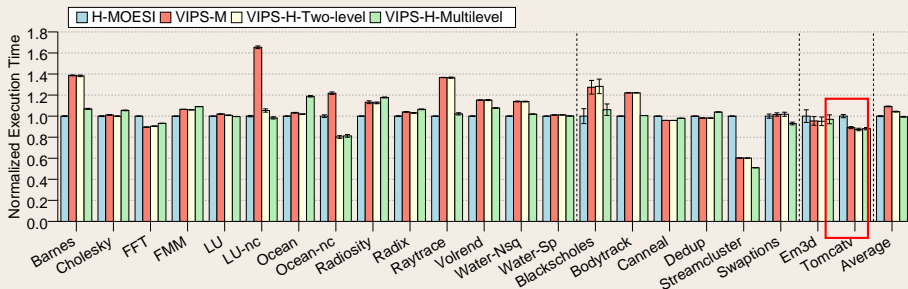
Controller	H-MOESI			H-Token			VIPS-H		
	States Tot./Base	Bitmap bits	Total bits	States Tot./Base	Tokens bits	Total bits	States Tot./Base	P/S bit	Total bits
L1 cache	16 / 5	0	3	16 / 5	7	10	9 / 3	1	3
L2 cache	59 / 13	16	20	6 / 4	7	9	5 / 3	1	3
L3 cache	13 / 4	4	6	3 / 2	7	8	4 / 3	1	3
Total cost	844KB		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

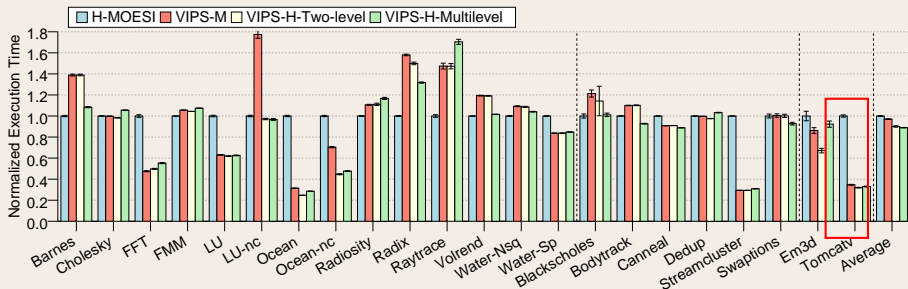
## 4x4 CLUSTERED SYSTEM



# 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

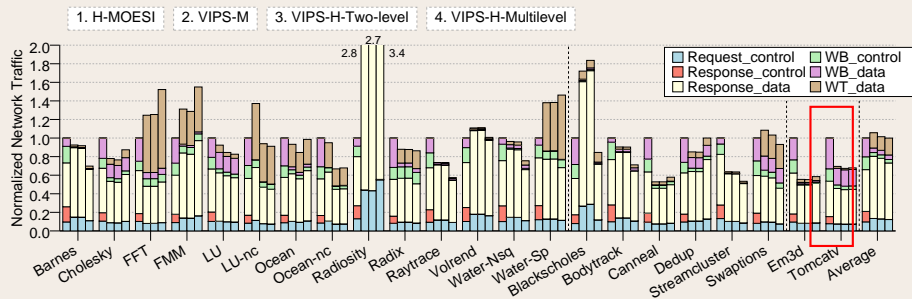
## 16x4 CLUSTERED SYSTEM



## 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)

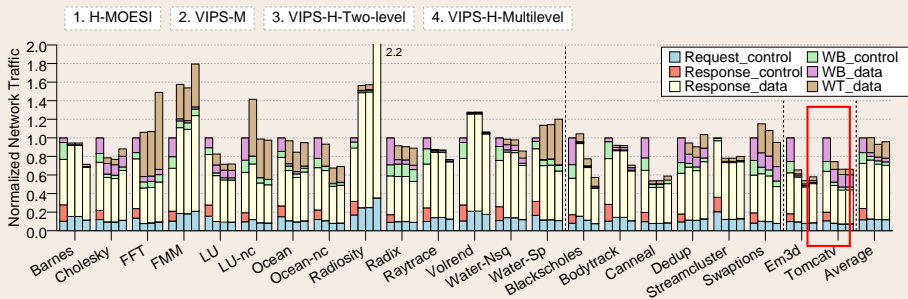
## 4x4 CLUSTERED SYSTEM



## 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)

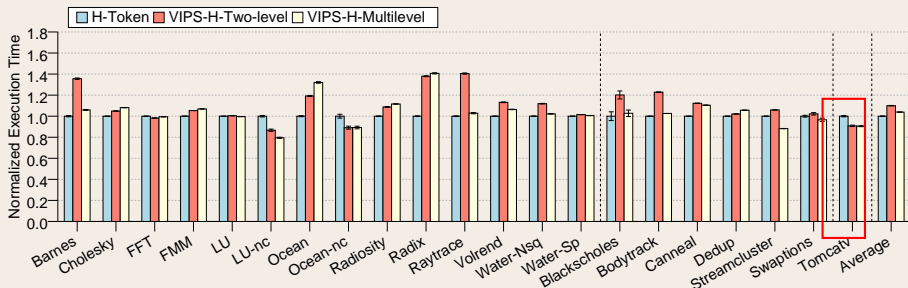
## 16x4 CLUSTERED SYSTEM



# 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

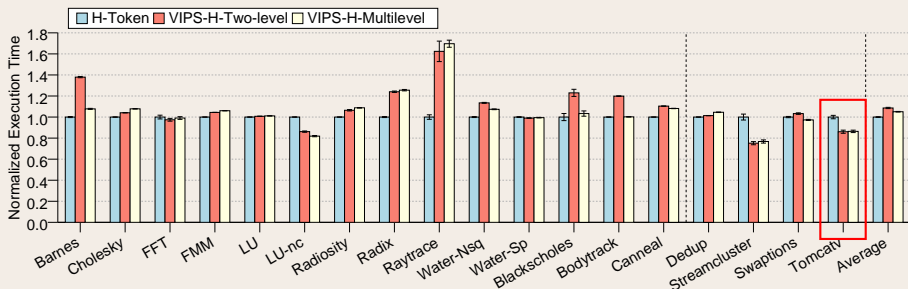
## 4x4 CLUSTERED SYSTEM



# 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

## 16x4 CLUSTERED SYSTEM

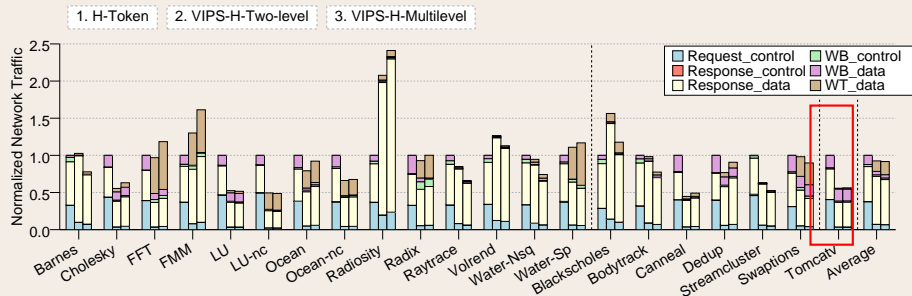




# COMPARISON TO H-TOKEN: NETWORK TRAFFIC

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

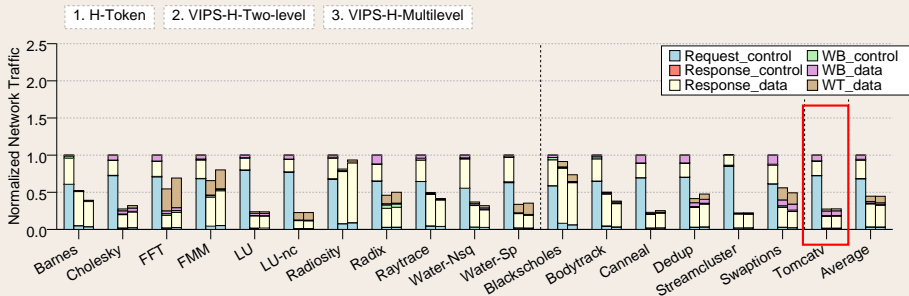
## 4x4 CLUSTERED SYSTEM



# COMPARISON TO H-TOKEN: NETWORK TRAFFIC

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

## 16x4 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  
⇒ **Complex**
  - Directory to track copies ⇒ **Non-scalable**
  - Indirection ⇒ **Inefficient**

# SIMPLE CACHE COHERENCE PROTOCOLS

- Simple cache coherence: **VIPS-M** [Ros & Kaxiras, PACT'12]
  - Strictly request-response  $\Rightarrow$  **Simple**
  - Coherence distributed across cores  $\Rightarrow$  **Scalable**
  - No directory  $\Rightarrow$  **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  $\Rightarrow$  **Simple**
  - Coherence distributed across cores  $\Rightarrow$  **Scalable**
  - No directory  $\Rightarrow$  **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;  
SIGNAL(cond);
```

```
WAIT(cond);  
$r1 = X;
```

# SIMPLE CACHE COHERENCE PROTOCOLS

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

**SD**

## EXAMPLE OF DRF CODE

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

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

```
WAIT(cond);  
$r1 = X;
```



# SIMPLE CACHE COHERENCE PROTOCOLS

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

## EXAMPLE OF DRF CODE

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

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

```
WAIT(cond);
```

```
$r1 = X; SI
```

# SIMPLE CACHE COHERENCE PROTOCOLS

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

## EXAMPLE OF DRF CODE

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

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

```
WAIT(cond);  
$r1 = X;
```

# THE PROBLEM OF THE DATA RACES

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

## EXAMPLE OF DRF CODE

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

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

```
WAIT(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;  
cond = 1;
```

```
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

## EXAMPLE OF DRF CODE

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

```
X = 1;  
cond = 1;
```

```
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  $\Rightarrow$  spin on last level cache (LLC)
    - Increases network traffic and LLC accesses  $\Rightarrow$  energy

## EXAMPLE OF DRF CODE

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

```
X = 1;  
cond = 1;
```

```
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  $\Rightarrow$  spin on last level cache (LLC)
    - Increases network traffic and LLC accesses  $\Rightarrow$  energy
- **VIPS-M** solution
  - $\Rightarrow$  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;
```

```
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  $\Rightarrow$  spin on last level cache (LLC)
    - Increases network traffic and LLC accesses  $\Rightarrow$  energy
- **VIPS-M** solution
  - $\Rightarrow$  Exponential back-off
  - 😊 Reduces SI, network traffic, and LLC accesses
  - ☹ Slows down propagation of writes

## EXAMPLE OF DRF CODE

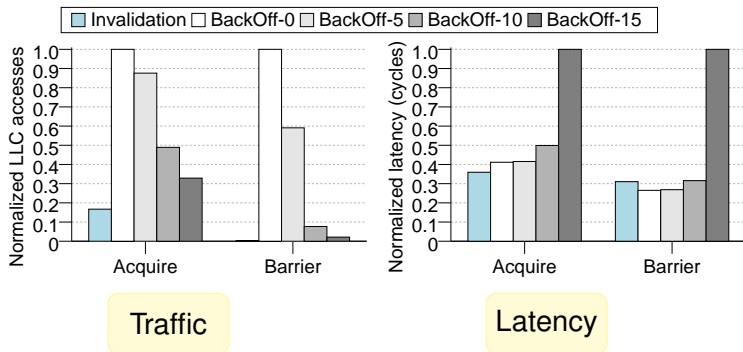
```
/* Initially X = 0 */
```

```
X = 1;  
cond = 1;
```

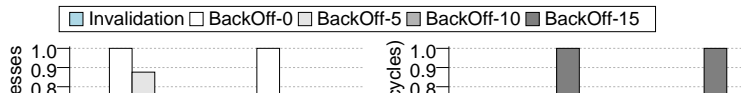
```
while(!cond);  
$r1 = X;
```



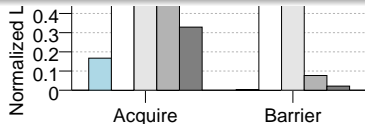
## ENERGY-PERFORMANCE TRADE-OFF



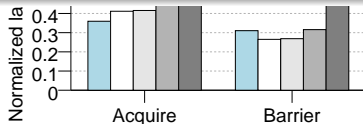
## ENERGY-PERFORMANCE TRADE-OFF



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

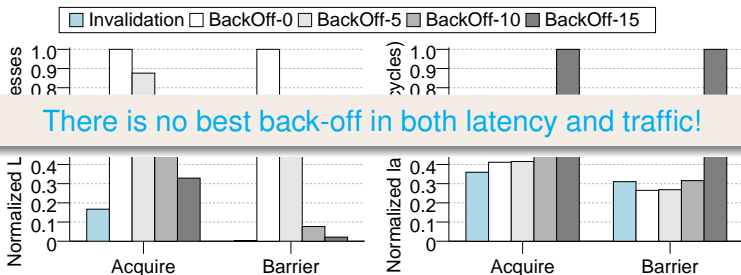


Traffic



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

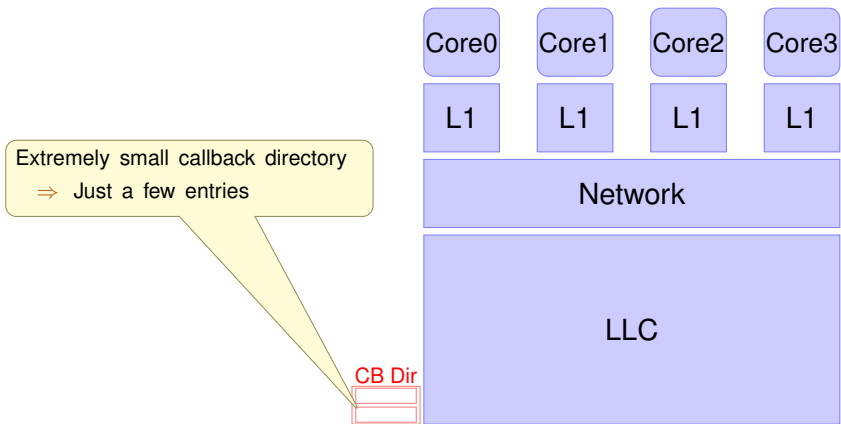
# CALLBACKS

- 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

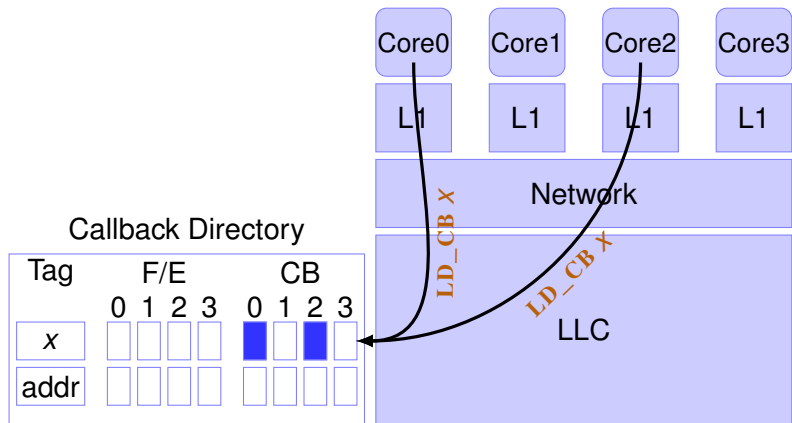
- Tag: word address
- F/E: Full/Empty bits per core
  - Full: It may be a new value to consume
  - Empty: There is no new value
- CB: Callback bits per core
  - A callback is blocked waiting for a new value

Callback Directory

Tag	F/E				CB			
	0	1	2	3	0	1	2	3
x	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
addr	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

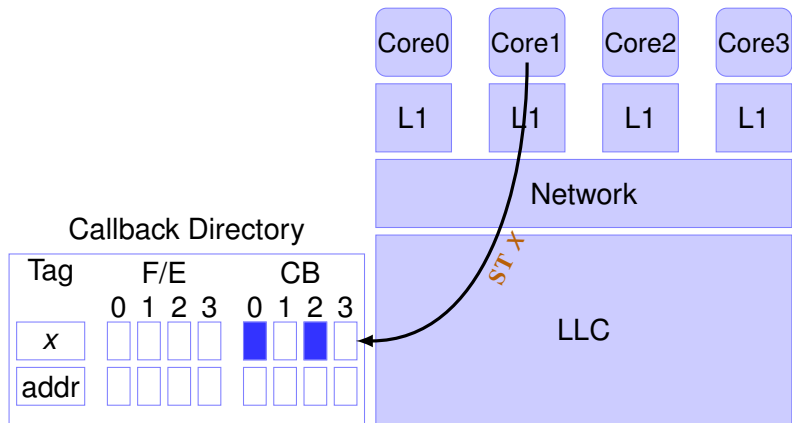
LLC

## CALLBACK EXAMPLE

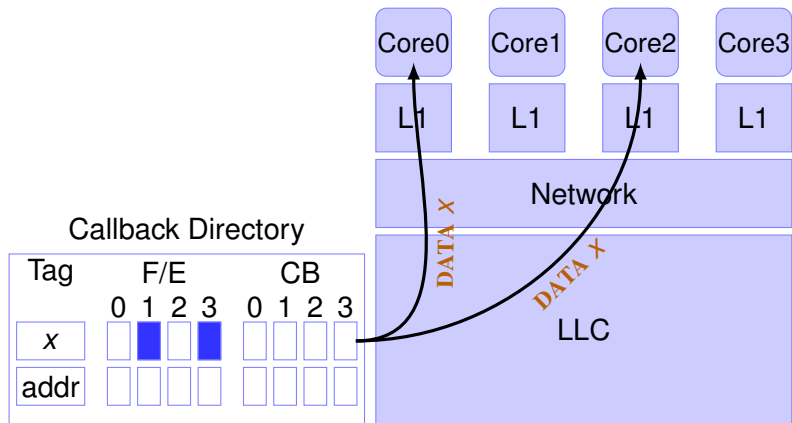




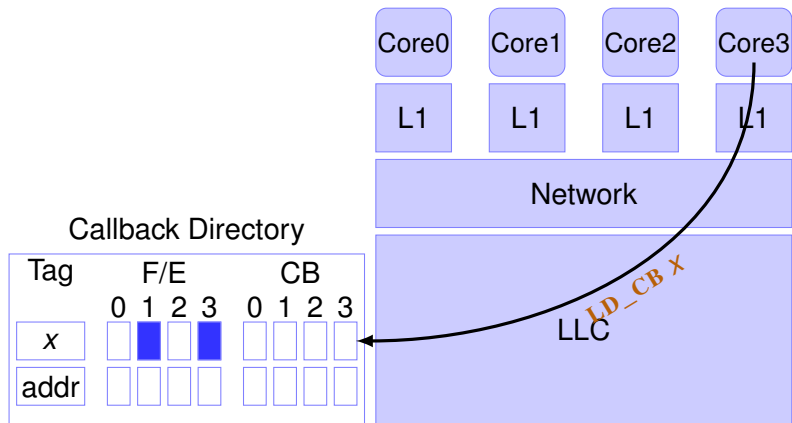
## CALLBACK EXAMPLE



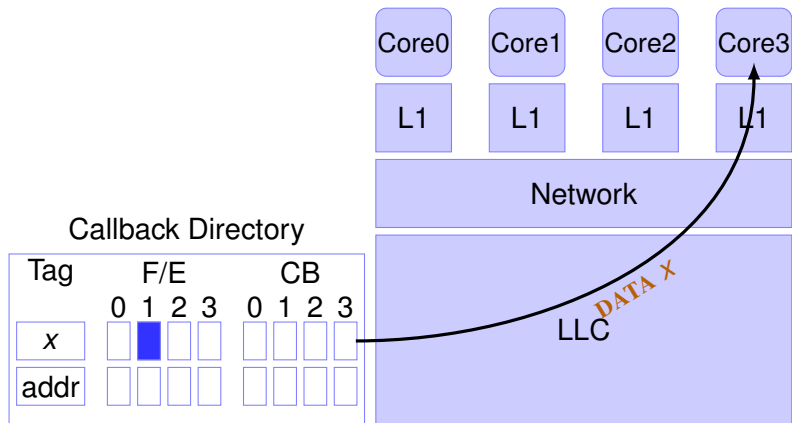
## CALLBACK EXAMPLE



## CALLBACK EXAMPLE



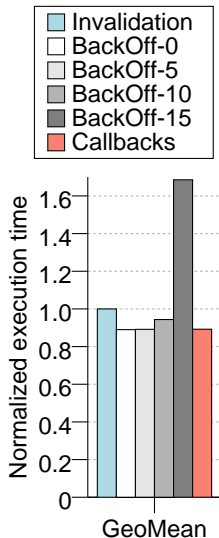
## CALLBACK EXAMPLE



# 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**



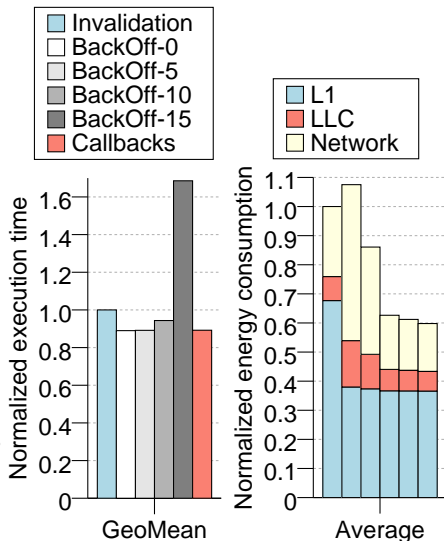
## EXECUTION TIME AND ENERGY CONSUMPTION

## Execution time

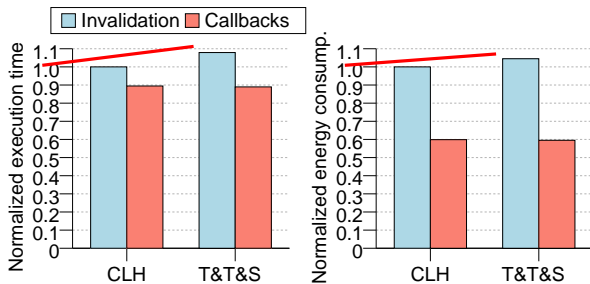
- As good as the best **BACKOFF** case
- 5% better than **BACKOFF-10** (**VIPS-M**)
- 11% better than **INVALIDATION**

## Energy consumption

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

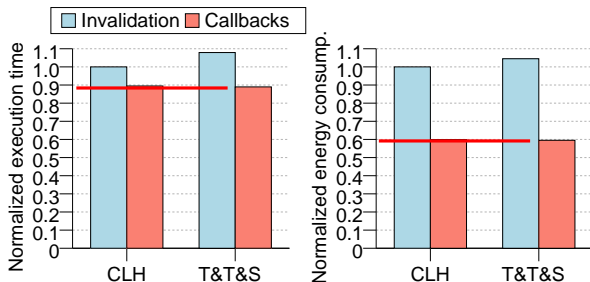


## TATAS vs. CLH



# TATAS vs. CLH

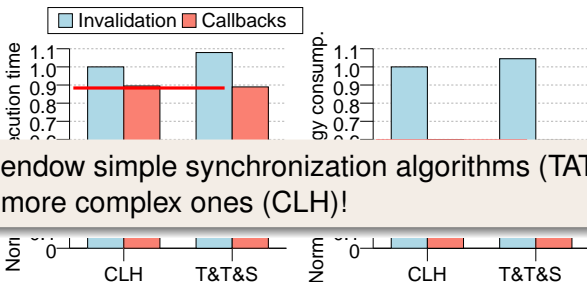
- T&T&S + Callbacks allows only one of the threads to race for acquiring the lock
- T&T&S + Callbacks provides fairness





# 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  
⇒ 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

- **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

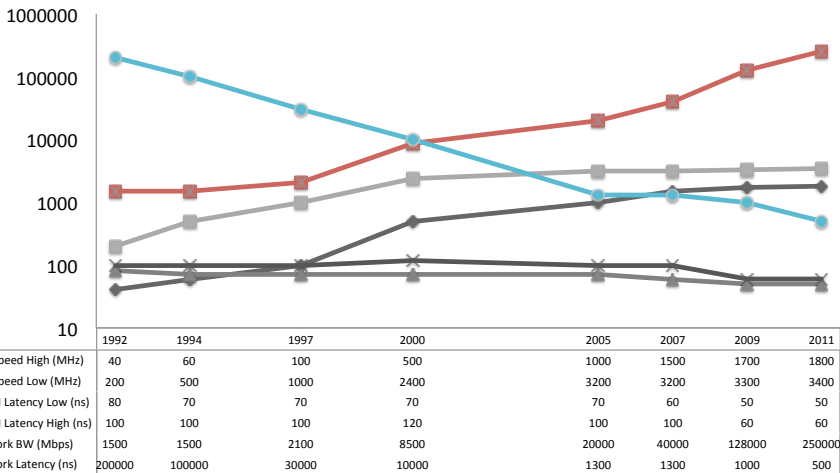
## ARGO

- **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?

CPU, DRAM and Network Trends



# 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

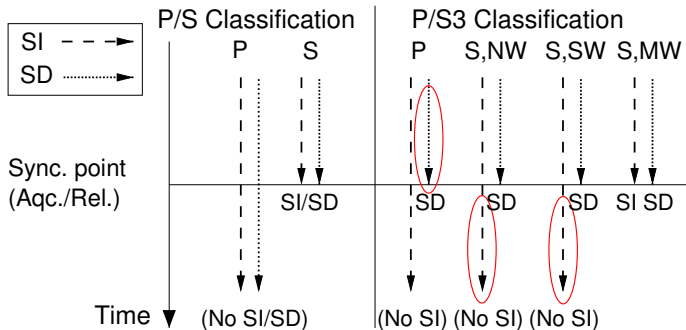
- Modified VIPS: SI & SD
  - Strictly request response for DRF accesses
- Pyxis classification directories cached at nodes
  - NO message handlers to classify pages and propagate classification changes
  - Requestors are responsible to update classification at remote nodes (P→S, requestor updates private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



# CARINA & PYXIS: COHERENCE & DIRECTORIES

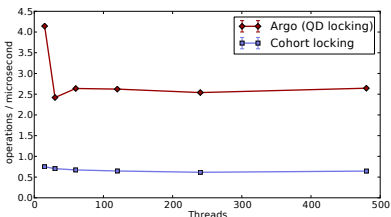
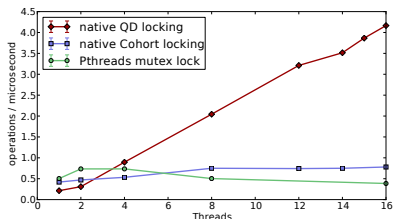
## ● Classification:

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



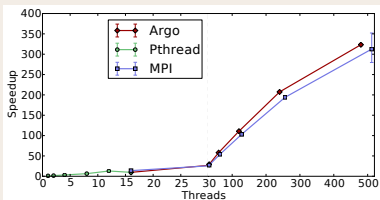
# VELA: ARGO'S SYNCHRONIZATION SYSTEM

- The trouble with distributed **critical 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

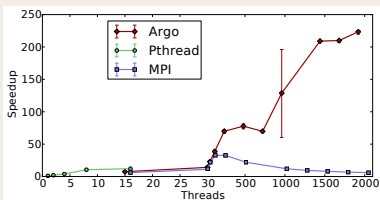


## RESULTS

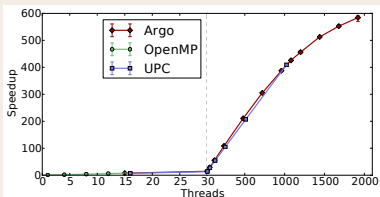
## NBODY



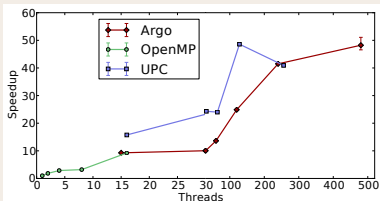
## BLACKSCHOLES



## 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<sub>1</sub>-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

19 de febrero de 2016

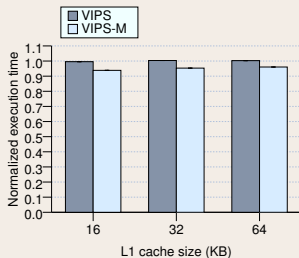
# SIMULATION ENVIRONMENT

- 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.

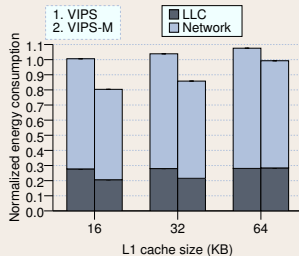
Protocol	Invalidations	Directory	Indirection	L1 base states
Hammer	Broadcast	None	Yes	5 (MOESI)
Directory	Multicast	Full-map	Yes	4 (MESI)
Write-Through	Multicast	Full-map	Only write misses	2 (VI)
VIPS	Multicast	Full-map	Only for write misses	2 (VI)
VIPS-M	None	None	No	2 (VI)

## EVALUATION

## PERFORMANCE 16KB-64KB L1



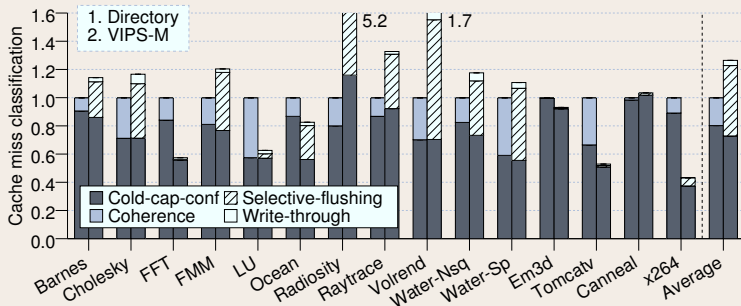
## ENERGY 16KB-64KB L1



# 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

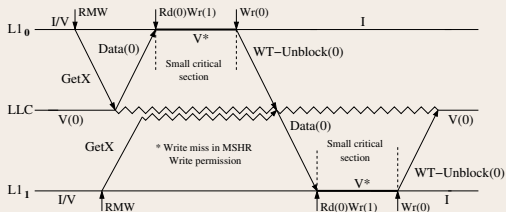




# VIPS-M

- 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