

VIPS: SIMPLE, EFFICIENT, AND SCALABLE CACHE COHERENCE

Alberto Ros¹ **Stefanos Kaxiras**²
Kostis Sagonas² Mahdad Davari²
Magnus Norgen² David Klaftenegger²

¹Universidad de Murcia
aros@dittec.um.es

²Uppsala University

Dec 17, 2015



MOTIVATION

- Cache coherence protocols ease **programming**
- Coherence **overhead** is an important issue
- But, coherence is sporadically needed
 - Why pay always?

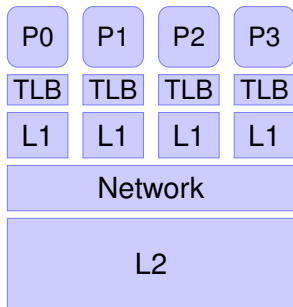
MOTIVATION

- Cache coherence protocols ease **programming**
- Coherence **overhead** is an important issue
- But, coherence is sporadically needed
 - Why pay always?
- Our goal → **Simplify coherence**
 - And enforce it only when needed

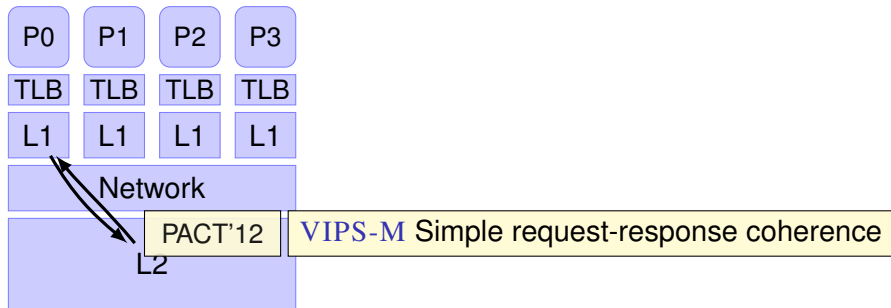
MOTIVATION

- Cache coherence protocols ease **programming**
- Coherence **overhead** is an important issue
- But, coherence is sporadically needed
 - Why pay always?
- Our goal → **Simplify coherence**
 - And enforce it only when needed
- How? **VIPS** family of cache coherence protocols
 - Simple, Efficient, Scalable

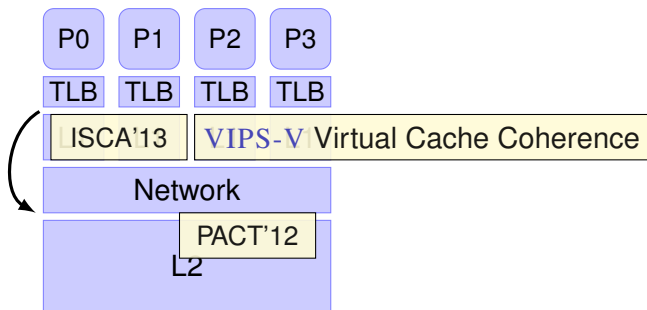
OVERVIEW



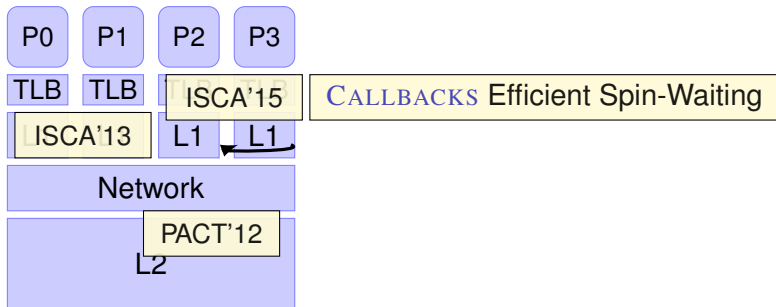
OVERVIEW



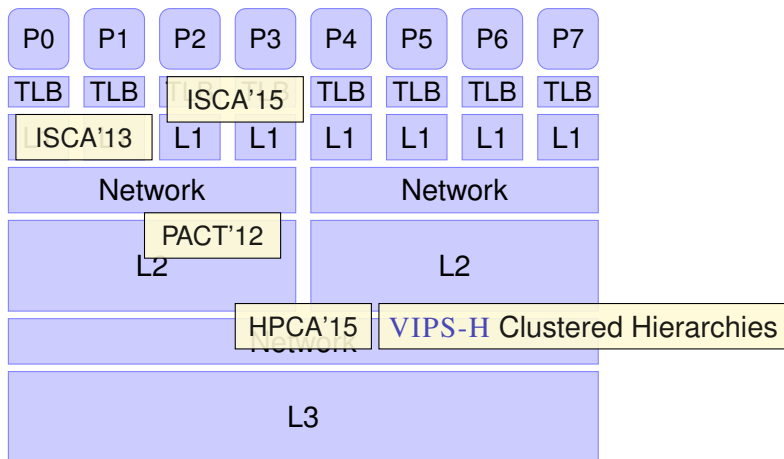
OVERVIEW



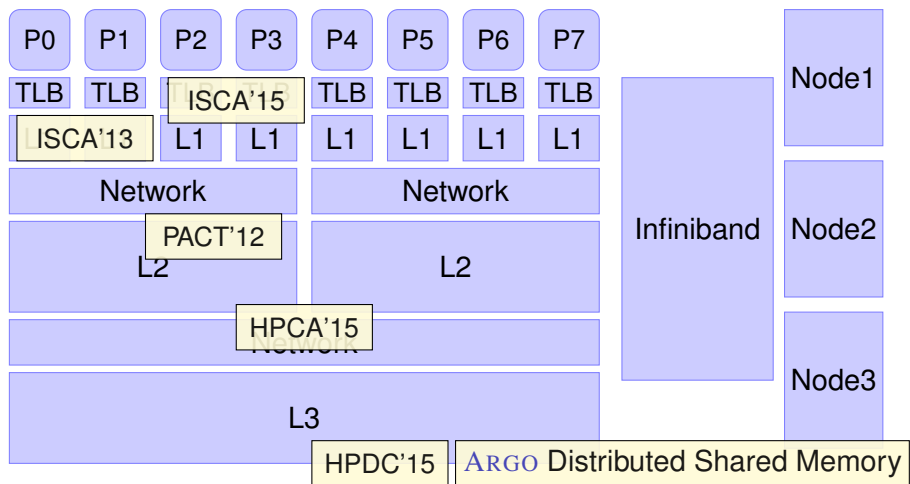
OVERVIEW



OVERVIEW



OVERVIEW



OUTLINE

OUTLINE

MOTIVATION

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

MOTIVATION

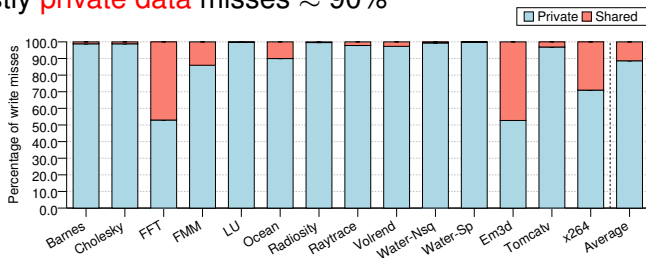
- **Write-through** protocols are simple
 - Only **Valid** and **Invalid** states
 - But they are not efficient because of **write misses**
- Which write misses?

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
- 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

VIPS PROPERTIES

- 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...

VIPS-M: SELF-INVALIDATION

- We provide **sequential consistency** for **DRF** programs
- Self-Invalidation of **shared** data from L1s
 - Selective Flush (SF) upon synchronization points
 - We eliminate **invalidations**
 - **The directory is gone!**
- Multiple writers allowed for **shared** data
 - Self-downgrade
 - No need to request write permission
 - Write-through of *diffs*

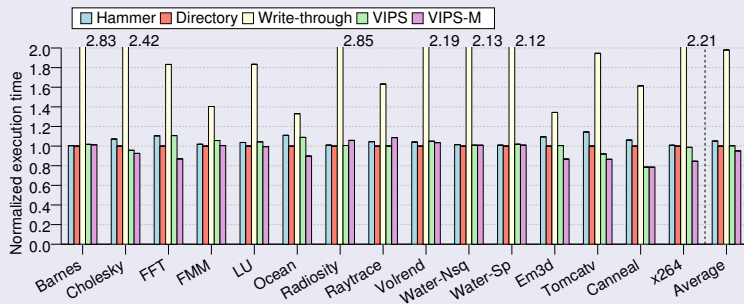
VIPS-M PROPERTIES

- 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

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

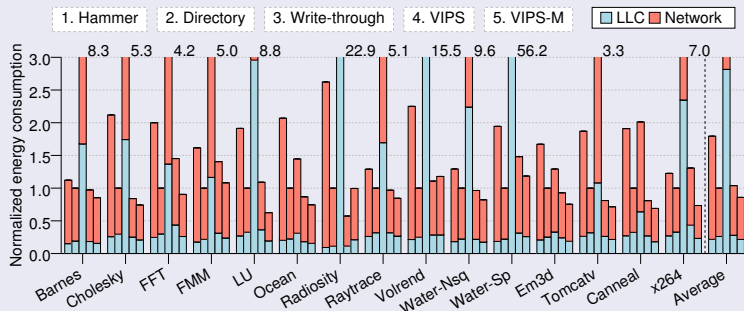
NORMALIZED EXECUTION TIME W.R.T. DIRECTORY



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

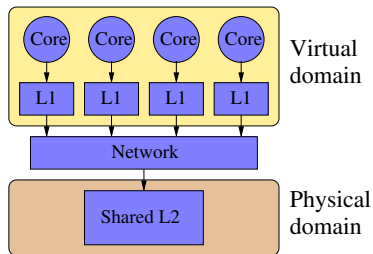


OUTLINE

- What is virtual-cache coherence?
 - Keeping cache coherence in a system with **virtual caches**

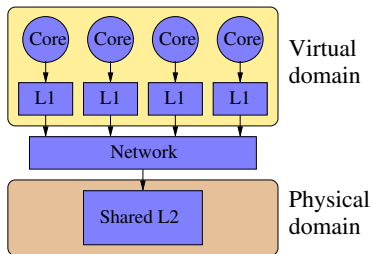
INTRODUCTION

- What is virtual-cache coherence?
 - Keeping cache coherence in a system with **virtual caches**
- Coherence is maintained for physical addresses (e.g., shared cache)



INTRODUCTION

- What is virtual-cache coherence?
 - Keeping cache coherence in a system with **virtual caches**
- Coherence is maintained for physical addresses (e.g., shared cache)

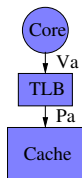


CONTRIBUTION

Simple and efficient approach that supports virtual caches in a cache coherence multicore system, thus saving most of the energy consumed by the TLBs

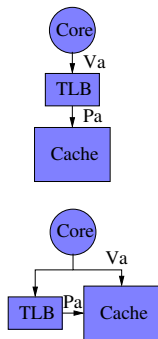
VIRTUAL VS. PHYSICAL CACHES

- Simple: Physically-indexed, physically-tagged (PIPT) caches
 - Address translation before accessing the cache
 - BUT: **high latency** and **high energy** consumption due to TLB accesses



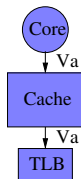
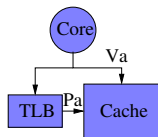
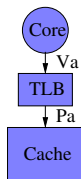
VIRTUAL VS. PHYSICAL CACHES

- Simple: Physically-indexed, physically-tagged (PIPT) caches
 - Address translation before accessing the cache
 - BUT: **high latency** and **high energy** consumption due to TLB accesses
- Performance: Virtually-indexed, physically-tagged (VIPT) caches
 - Translation before comparing the tags
 - TLB and cache accessed in parallel → latency OK
 - BUT STILL: **high energy** consumption



VIRTUAL VS. PHYSICAL CACHES

- Simple: Physically-indexed, physically-tagged (PIPT) caches
 - Address translation before accessing the cache
 - BUT: **high latency** and **high energy** consumption due to TLB accesses
- Performance: Virtually-indexed, physically-tagged (VIPT) caches
 - Translation before comparing the tags
 - TLB and cache accessed in parallel → latency OK
 - BUT STILL: **high energy** consumption
- Efficient: Virtually-indexed, virtually-tagged (VIVT) caches
 - No TLB translation required on cache hits
 - **NO extra latency or energy** on cache hits
 - Larger TLBs, shared TLBs
 - Problem: **synonyms**



VIRTUAL CACHES IN UNI- AND MULTI-PROCESSORS

- Synonyms: Different virtual addresses mapping to the same physical address
 - Address mapping changes or sharing among processes
- **IN VIVT CACHES:** Multiple copies of the same (physical) block in cache → inconsistency
 - Hardware solutions (**complex, expensive**): Upon a miss check if there are synonyms
 - Cache search: Looks in all possible sets
- **IN MULTIPROCESSORS:** **Reverse translation** for messages going from the physical to the virtual domain
 - Reverse map (R-tag memory) [Goodman, ASPLOS'87]
 - Hardware and memory requirements, and design **complexity**

AVOIDING REVERSE TRANSLATION

- We address this problem by focusing on the coherence protocol
- When is reverse translation performed?
 - For every coherence message sent from the physical domain (shared cache) to the virtual domain (private cache)
 - In traditional coherence protocols:
 - **Invalidations**, **downgrades**, and **forwardings**: Not expected by the cache controller (no MSHR entry)
 - **Data** and **acks**: expected by the cache controller (MSHR entry)

AVOIDING REVERSE TRANSLATION

- We address this problem by focusing on the coherence protocol
- When is reverse translation performed?
 - For every coherence message sent from the physical domain (shared cache) to the virtual domain (private cache)
 - In traditional coherence protocols:
 - **Invalidations**, **downgrades**, and **forwardings**: Not expected by the cache controller (no MSHR entry)
 - **Data** and **acks**: expected by the cache controller (MSHR entry)

AVOIDING REVERSE TRANSLATION

- We address this problem by focusing on the coherence protocol

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

- **Data** and **acks**: expected by the cache controller (MSHR entry)

AVOIDING REVERSE TRANSLATION

- We address this problem by focusing on the coherence protocol
- When is reverse translation performed?
 - For every coherence message sent from the physical domain (shared cache) to the virtual domain (private cache)
 - In traditional coherence protocols:
 - **Invalidations**, **downgrades**, and **forwardings**: Not expected by the cache controller (no MSHR entry)
 - **Data** and **acks**: expected by the cache controller (MSHR entry)
- Can coherence protocols satisfy the previous condition while being **efficient**?

AVOIDING REVERSE TRANSLATION

- We address this problem by focusing on the coherence protocol
- When is reverse translation performed?
 - For every coherence message sent from the physical domain (shared cache) to the virtual domain (private cache)
 - In traditional coherence protocols:
 - **Invalidations**, **downgrades**, and **forwardings**: Not expected by the cache controller (no MSHR entry)
 - **Data** and **acks**: expected by the cache controller (MSHR entry)
- Can coherence protocols satisfy the previous condition while being **efficient**?
 - Yes, VIPS-M!

VIPS-M IS THE RIGHT STUFF

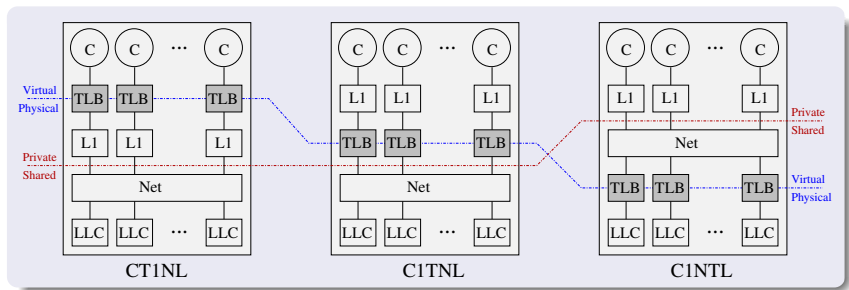
- Self-Invalidation eliminates directory invalidations
 - **No invalidations** issued from the LLC to the L1s
- Write-throughs keep data clean in the L1 caches
 - **No downgrades** issued from the LLC to the L1s
- Write-throughs keep data updated in the LLC caches
 - **No forwardings** issued from the LLC to the L1s
 - Indirection completely removed

VIPS-M IS THE RIGHT STUFF

- Self-Invalidation eliminates directory invalidations
 - **No invalidations** issued from the LLC to the L1s
- Write-throughs keep data clean in the L1 caches
 - **No downgrades** issued from the LLC to the L1s
- Write-throughs keep data updated in the LLC caches
 - **No forwardings** issued from the LLC to the L1s
 - Indirection completely removed

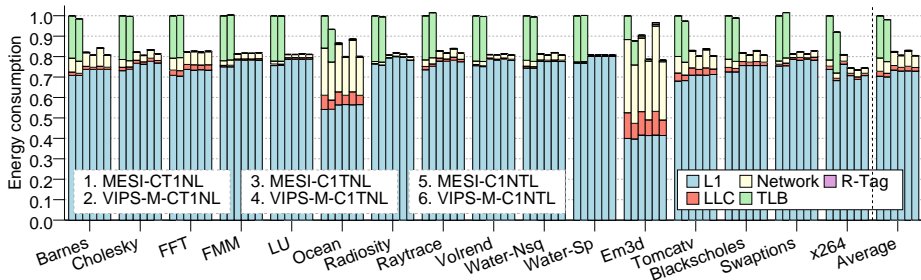
VIPS-M can work with virtual caches without requiring reverse translation and in the presence of synonyms

DESIGN CHOICES FOR TLB PLACEMENT



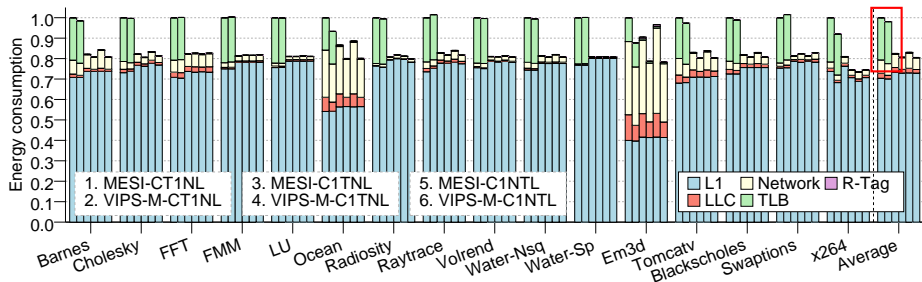
- CT₁NL: Physically-tagged L1 caches
- C₁TNL: Virtual L1 caches, private TLBs
- C₁NTL: Virtual L1 caches, shared TLBs

ENERGY CONSUMPTION



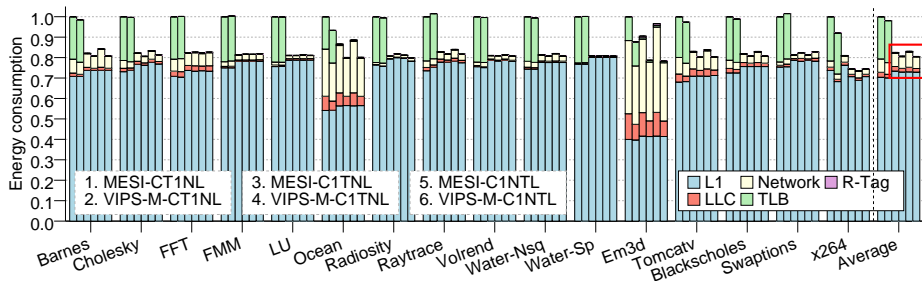
ENERGY CONSUMPTION

- Around 17% in energy reduction thanks to the use of virtual caches, mainly because of TLBs lookups



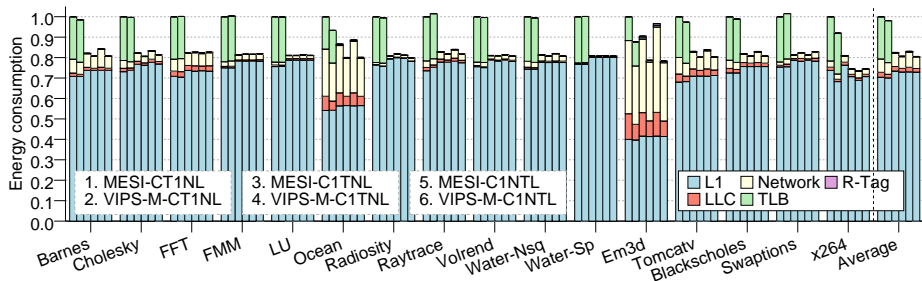
ENERGY CONSUMPTION

- Around 17% in energy reduction thanks to the use of virtual caches, mainly because of TLBs lookups
- VIPS-M keeps its advantage w.r.t. MESI (savings of 20% in total)



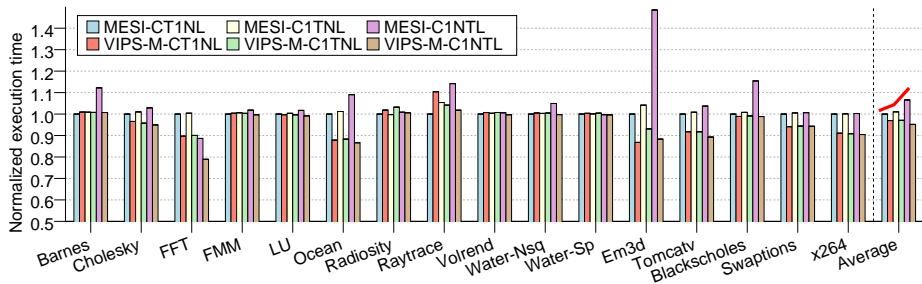
ENERGY CONSUMPTION

- Around 17% in energy reduction thanks to the use of virtual caches, mainly because of TLBs lookups
- VIPS-M keeps its advantage w.r.t. MESI (savings of 20% in total)
- The VIPS-M with **virtual caches** consume similar energy



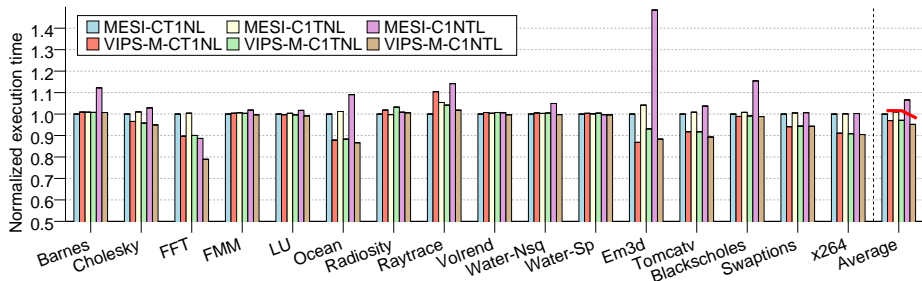
EXECUTION TIME

- Reverse translation is a problem for MESI protocols, especially for the shared TLB configuration



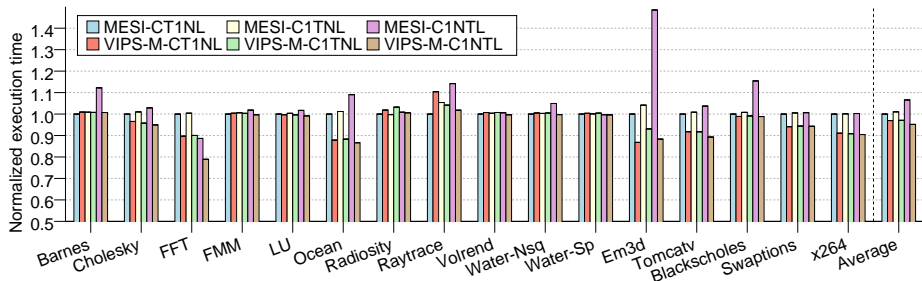
EXECUTION TIME

- Reverse translation is a problem for MESI protocols, especially for the shared TLB configuration
- VIPS-M obtains improvements by sharing the TLB



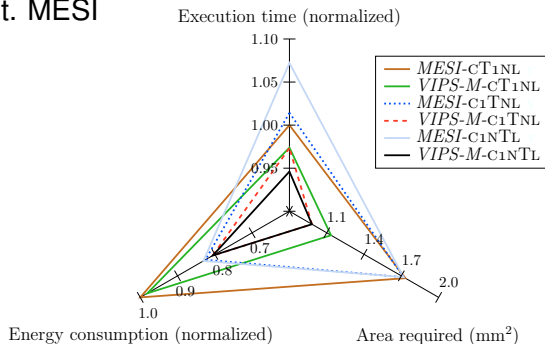
EXECUTION TIME

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



CONCLUSIONS

- Virtual cache coherence can be implemented **without reverse translations** and without increasing complexity
- Our approach obtains **execution time, energy, and area improvements** w.r.t. MESI



OUTLINE

MOTIVATION

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

- 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 */
```

<pre>X = 1; SIGNAL(cond);</pre>	<pre>WAIT(cond); \$r1 = X;</pre>
-------------------------------------	--------------------------------------

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

EXAMPLE OF DRF CODE

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

SD

```
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 */
```

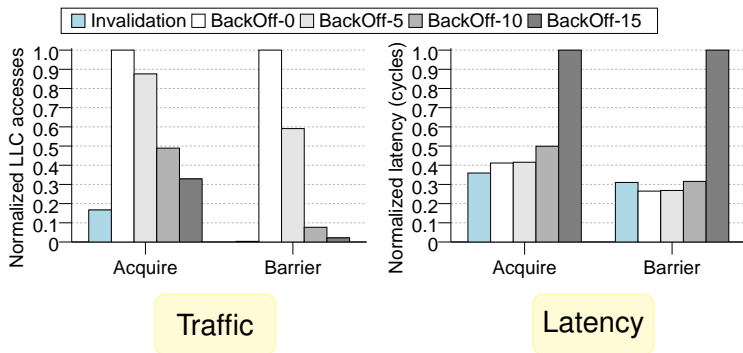
```
X = 1;  
cond = 1;
```

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

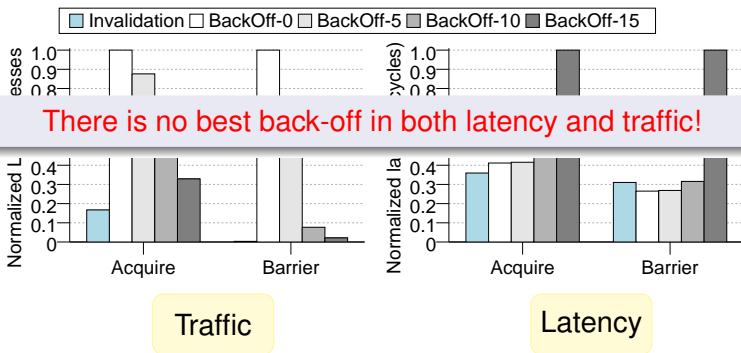


Energy-performance trade-off!

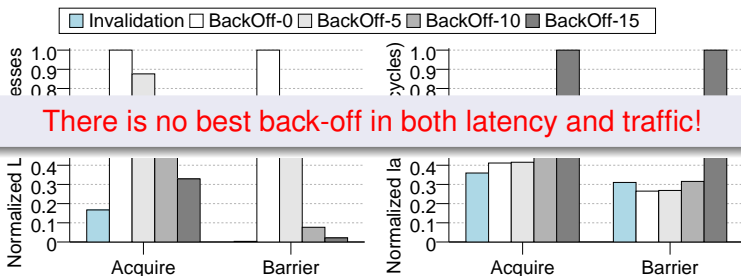
ENERGY-PERFORMANCE TRADE-OFF



ENERGY-PERFORMANCE TRADE-OFF



ENERGY-PERFORMANCE TRADE-OFF



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

THE CHALLENGE

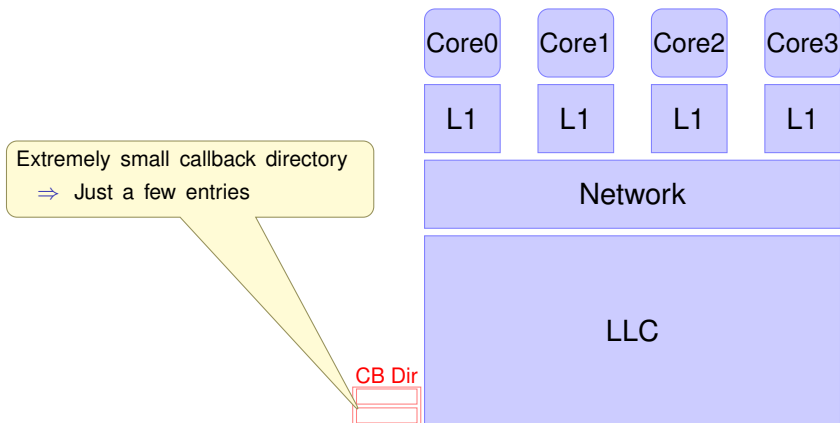
Fast and efficient write propagation...

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

- 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 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

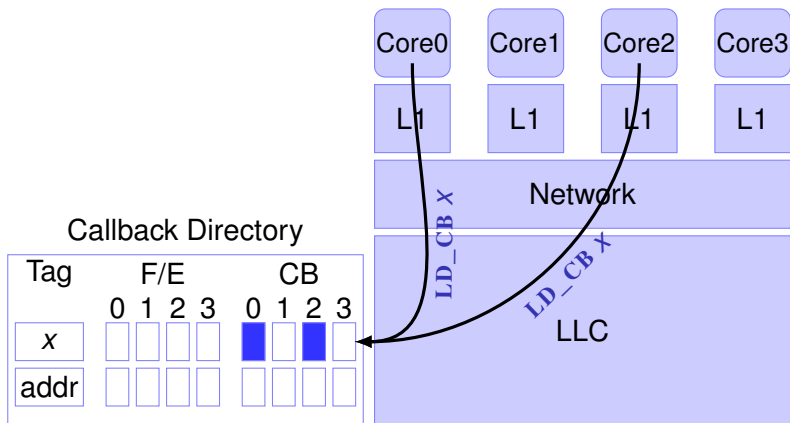
- Tag: word address
- F/E: Full/Empty bits per core
 - Full: It may be a new value to consume
 - Empty: There is no new value
- CB: Callback bits per core
 - A callback is blocked waiting for a new value

Callback Directory

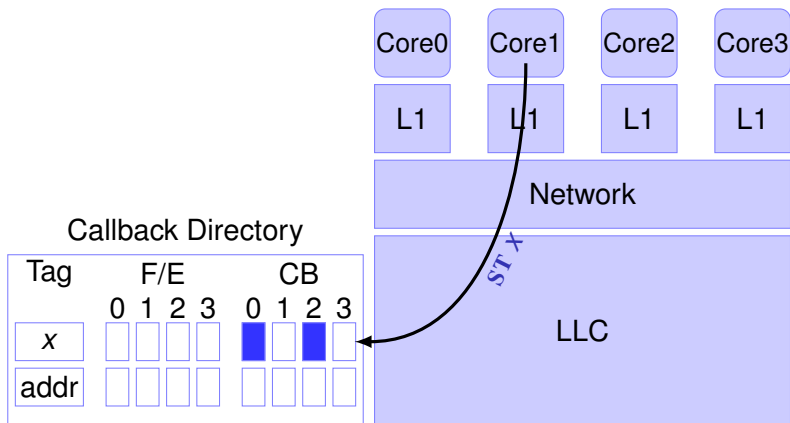
Tag	F/E				CB			
	0	1	2	3	0	1	2	3
x	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
addr	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>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

LLC

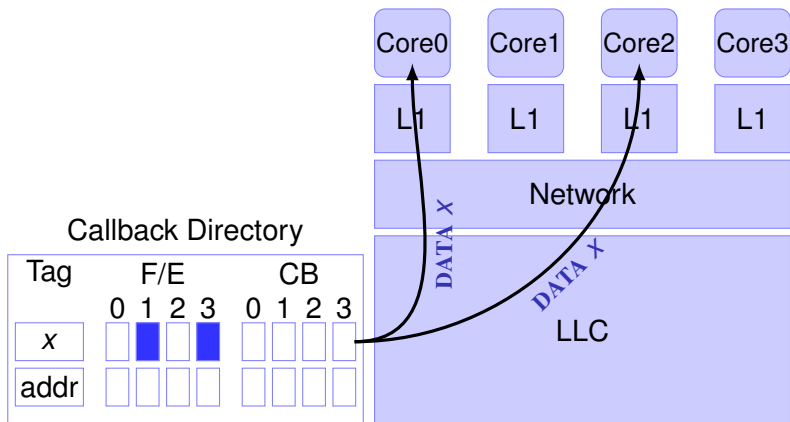
CALLBACK EXAMPLE



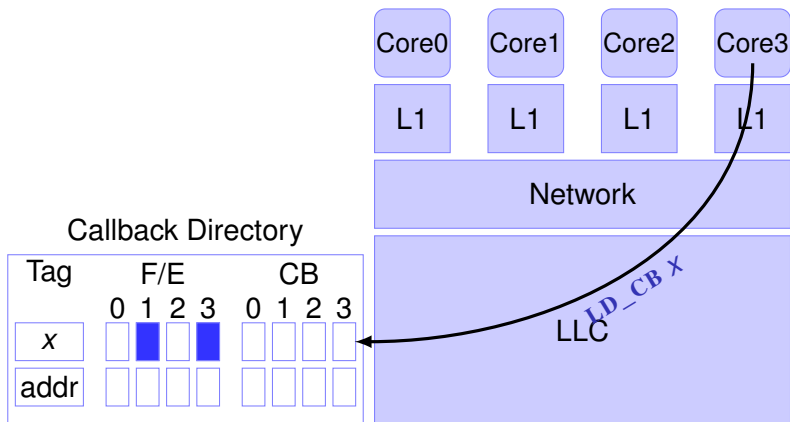
CALLBACK EXAMPLE



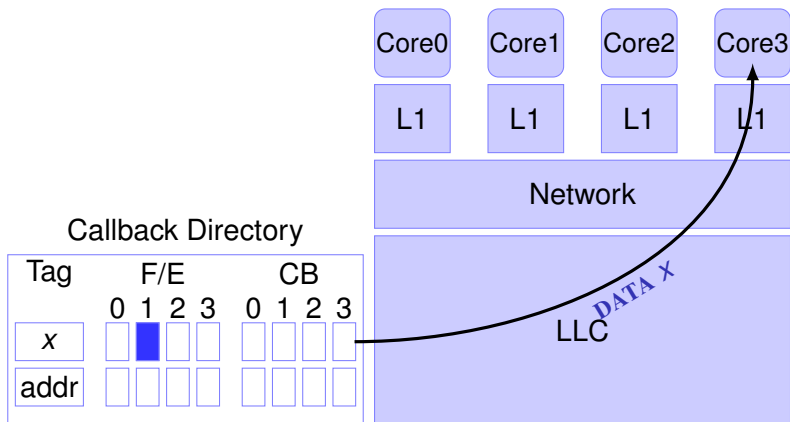
CALLBACK EXAMPLE



CALLBACK EXAMPLE



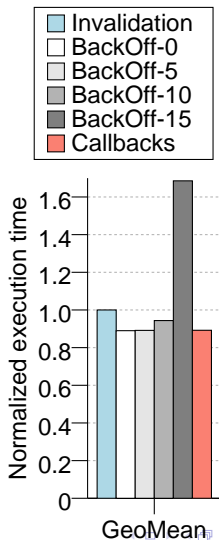
CALLBACK EXAMPLE



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**



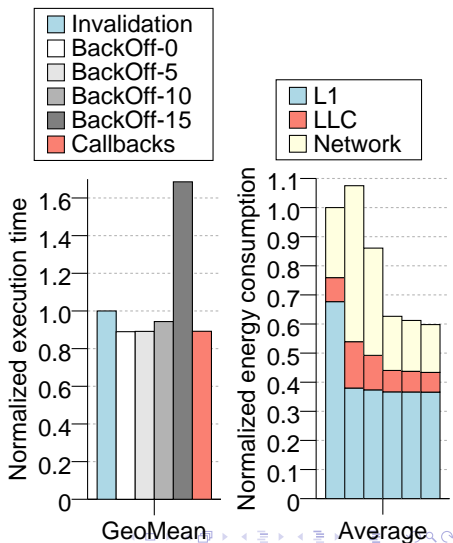
EXECUTION TIME AND ENERGY CONSUMPTION

Execution time

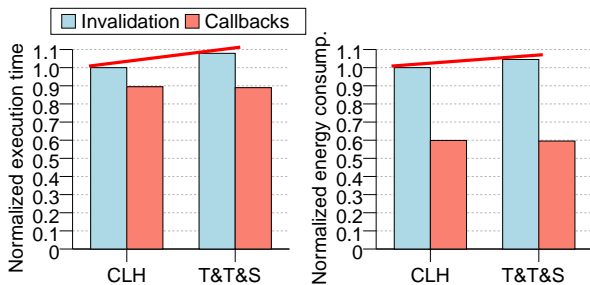
- As good as the best **BACKOFF** case
- 5% better than **BACKOFF-10** (**VIPS-M**)
- 11% better than **INVALIDATION**

Energy consumption

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

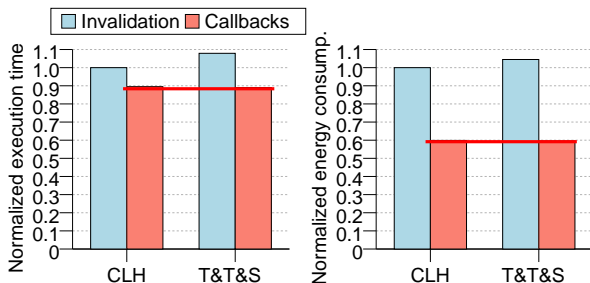


TATAS vs. CLH



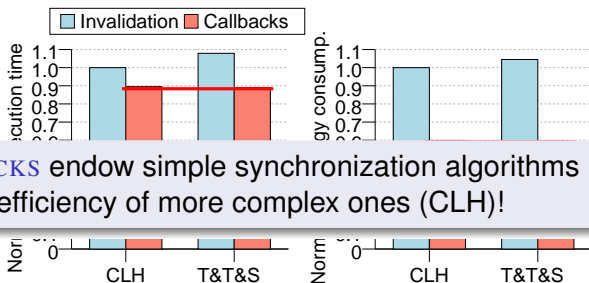
TATAS vs. CLH

- 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

OUTLINE

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.

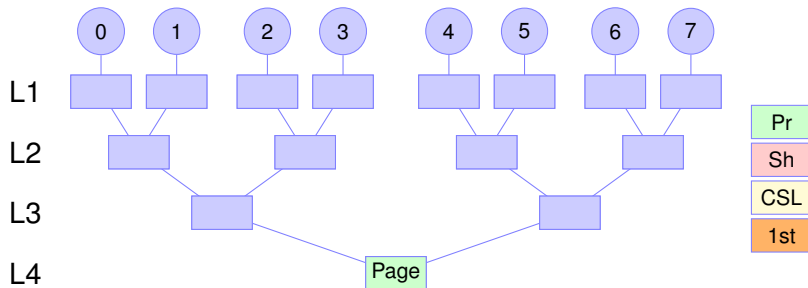
- Simplify the source of complexity: invalidation/downgrade
 - No write-invalidation \Rightarrow self-invalidation (SI) on synchronization points
 - No read-downgrade \Rightarrow self-downgrade (SD) on synchronization points
 - Provide sequential consistency for data-race-free (SC for DRF) applications

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

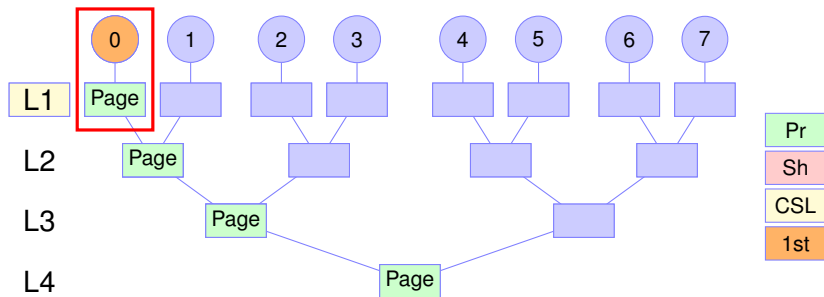
- We solve this problem by introducing the concept of **hierarchical P/S classification**
 - A block can be shared inside a cluster but be private outside
 - The level where this transition happens is the **common sharing level** (CSL)
 - Restrict SI/SD to shared blocks within a cluster
- **Result**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



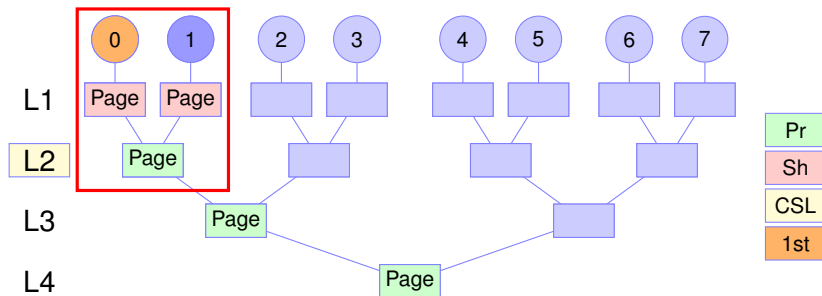
- SI/SD only for blocks in shared pages

COMMON SHARING LEVEL



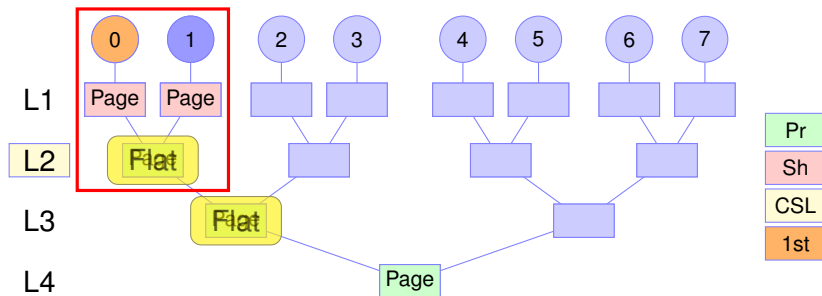
- SI/SD only for blocks in shared pages

COMMON SHARING LEVEL



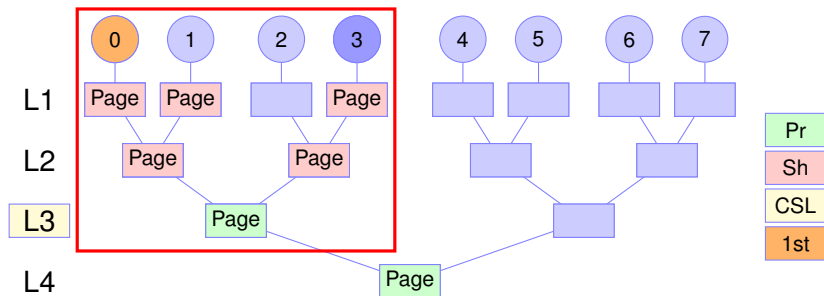
- SI/SD only for blocks in shared pages

COMMON SHARING LEVEL



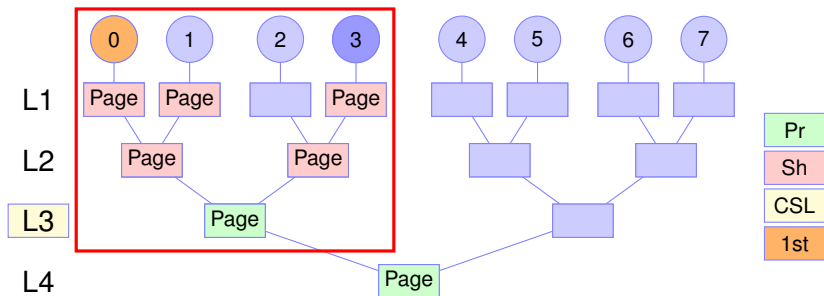
- SI/SD only for blocks in shared pages

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)

COMPLEXITY, COST, AND AREA

- H-MOESI: Hierarchical full-map
- VIPS-H: P/S bit

NUMBER OF STATES AND EXTRA BITS REQUIRED (16X4)

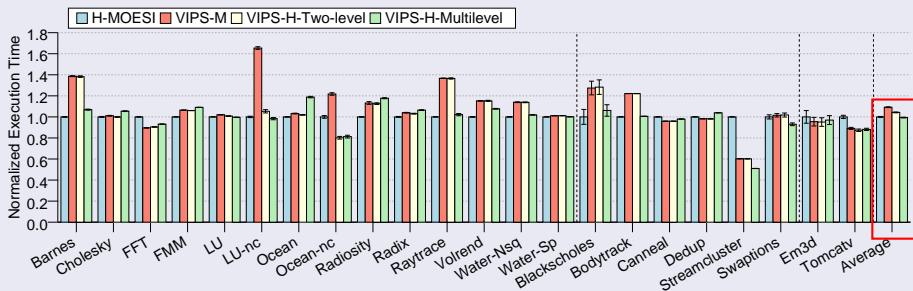
Controller	H-MOESI			VIPS-H		
	States Tot./Base	Bitmap bits	Total bits	States Tot./Base	P/S bit	Total bits
L1 cache	16 / 5	0	3	9 / 3	1	3
L2 cache	59 / 13	16	20	5 / 3	1	3
L3 cache	13 / 4	4	6	4 / 3	1	3
Total cost	844KB			204KB		

- **76%** memory reduction compared to H-MOESI

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

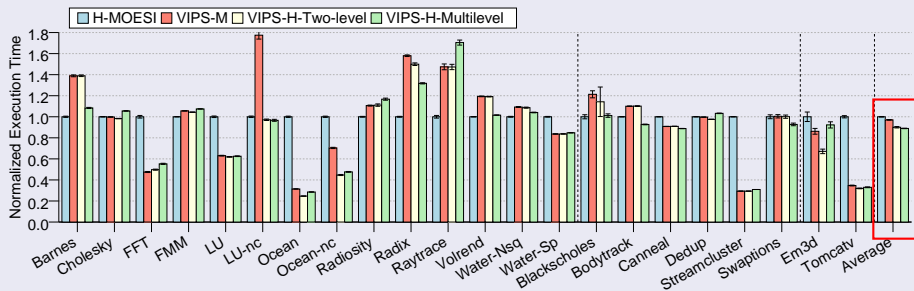
4x4 CLUSTERED SYSTEM



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

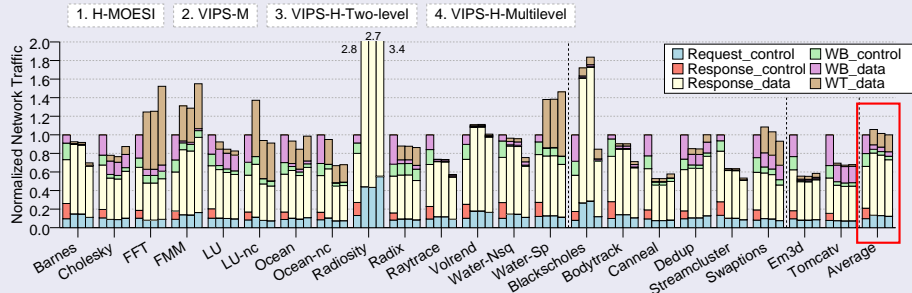
16x4 CLUSTERED SYSTEM



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)

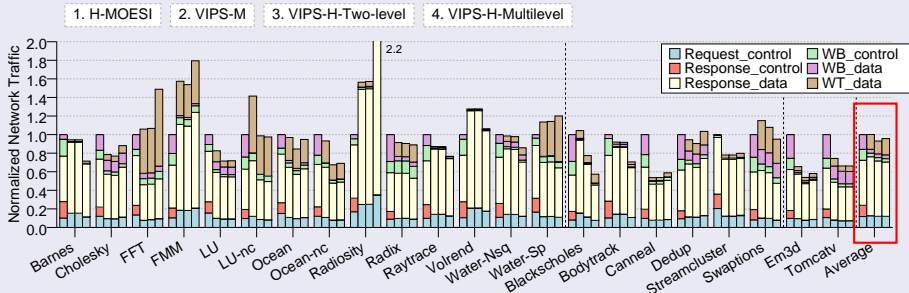
4x4 CLUSTERED SYSTEM



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



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

OUTLINE

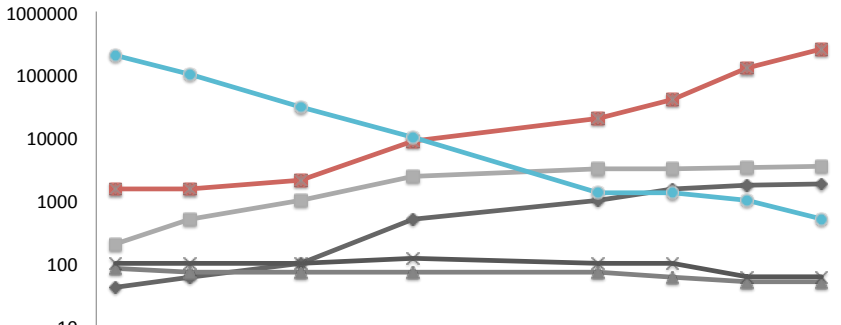
- VIPS-M \Rightarrow Self-Invalidation & Self-Downgrade
 - VIPS coherence is truly **distributed**.
 - Coherence decisions are taken independently without any inter-core interaction
 - \Rightarrow Simplifies whole system design
- Request-Response from the L1s to the LLC
 - No requests from LCC to L1s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

- 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?

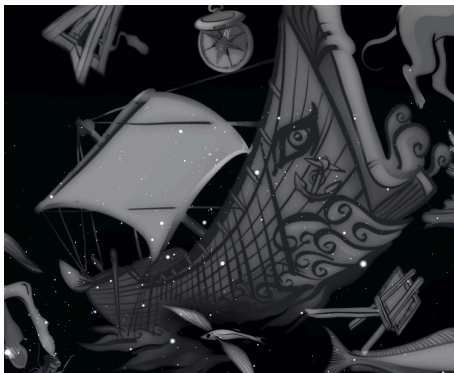
CPU, DRAM and Network Trends



	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

- 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



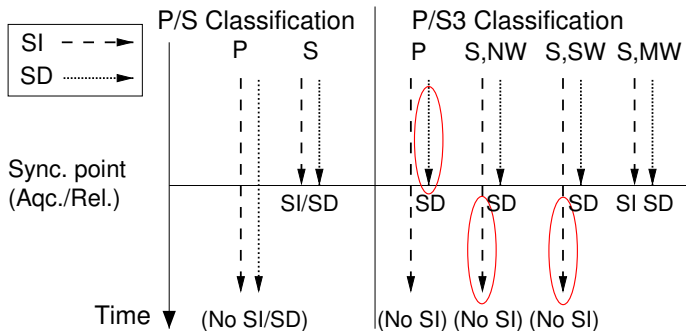
- **CARINA**: VIPS-DSM coherence
- **PYXIS**: Classification directories
- **VELA**: Hierarchical Queue Delegation Locking system

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)

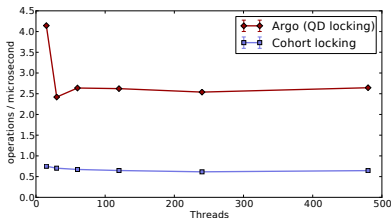
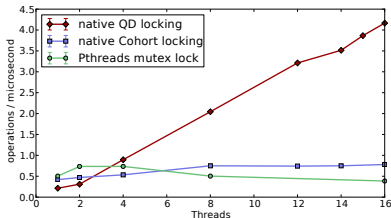
CARINA & PYXIS: COHERENCE & DIRECTORIES

- 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)



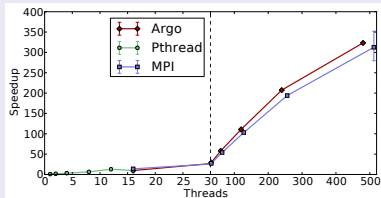
VELA: ARGO'S SYNCHRONIZATION SYSTEM

- The trouble with distributed **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

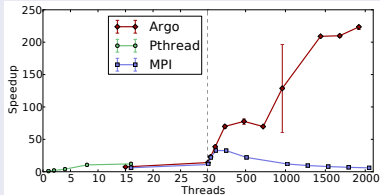


RESULTS

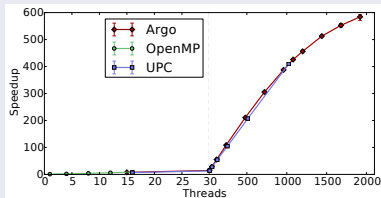
NBODY



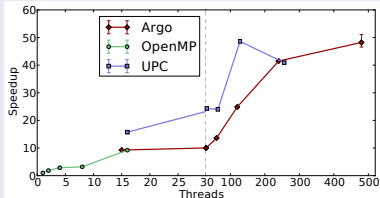
BLACKSCHOLES



EP CLASS D



CG CLASS C



OUTLINE

- 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]
 - Dir₁-SISD: Self-Contained Directories [PACT'15]
 - VIPS-G: CPU-GPU Coherence [TACO'16]

VIPS: SIMPLE, EFFICIENT, AND SCALABLE CACHE COHERENCE

Alberto Ros¹ **Stefanos Kaxiras**²
Kostis Sagonas² Mahdad Davari²
Magnus Norgen² David Klaftenegger²

¹Universidad de Murcia
aros@dittec.um.es

²Uppsala University

Dec 17, 2015



SIMULATION ENVIRONMENT

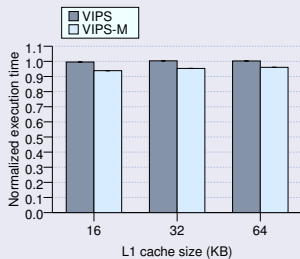
- SIMICS (functional simulation) + GEMS (memory timing) + GARNET (network)
- CACTI 6.5 for 32nm technology
- Simulated a **16-tile multicore**
 - 32KB 4-way I&D L1s, 8MB (512KB/bank) 16-way L2 (LLC)
 - 16-entry MSHRs with 1000-cycle timeout
- SPLASH-2, scientific, and PARSEC benchmarks.

Protocol	Invalidation	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)

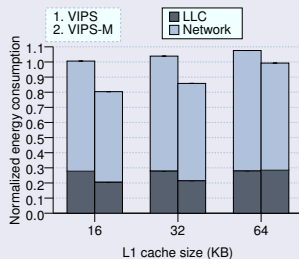
EVALUATION

L1 SENSITIVITY ANALYSIS

PERFORMANCE 16KB-64KB L1



ENERGY 16KB-64KB L1

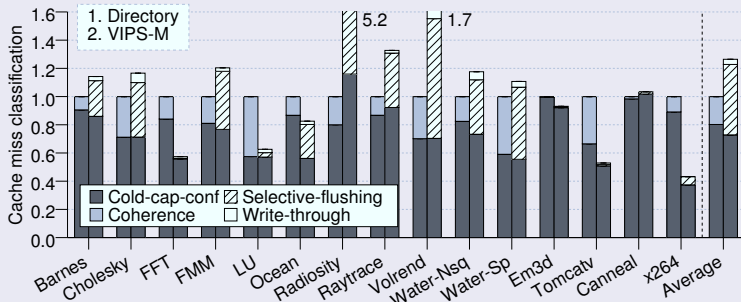


RESULTS

CACHE MISSES

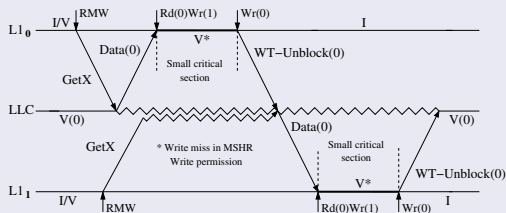
- Cold-cap-conf misses decrease due to the lack of write misses for DRF blocks
- Misses due to write throughs are not significant

CACHE MISSES NORMALIZED W.R.T. DIRECTORY



- 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