Hierarchical Private/Shared Classification: The Key to Simple and Efficient Coherence for Clustered Cache Hierarchies

Alberto Ros
Department of Computer Engineering
Universidad de Murcia, Spain
Email: aros@ditec.um.es

Mahdad Davari and Stefanos Kaxiras
Department of Information Technology
Uppsala University, Sweden
Email: {mahdad.davari,stefanos.kaxiras}@it.uu.se

***Abstract*—Hierarchical clustered cache designs are becoming an appealing alternative for multicores. Grouping cores and their caches in clusters reduces network congestion by localizing traffic among several hierarchical levels, potentially enabling much higher scalability. While such architectures can be formed recursively by replicating a base design pattern, keeping the whole hierarchy coherent requires more effort and consideration. The reason is that, in hierarchical coherence, even basic operations must be recursive. As a consequence, intermediate-level caches behave both as directories and as leaf caches. This leads to an explosion of states, protocol-races, and protocol complexity. While there have been previous efforts to extend directory-based coherence to hierarchical designs their increased complexity and verification cost is a serious impediment to their adoption.**

We aim to address these concerns by encapsulating all hierarchical complexity in a simple function: that of determining when a data block is shared entirely within a cluster (sub-tree of the hierarchy) and is private from the outside. This allows us to eliminate complex recursive operations that span the hierarchy and instead employ simple coherence mechanisms such as self-invalidation and write-through—now restricted to operate within the cluster where a data block is shared.

We examine two inclusivity options and discuss the relation of our approach to the recently proposed Hierarchical-Race-Free (HRF) memory models. Finally, comparisons to a hierarchical directory-based MOESI, VIPS-M, and TokenCMP protocols show that, despite its simplicity our approach results in competitive performance and decreased network traffic.

I. Introduction

Setting: In their road map to scalable on-chip cache coherence, Martin *et al.* [27] advocate hierarchical and clustered design techniques as natural methodology for future scalable systems to overcome two main scalability problems of coherence: storage and traffic. Storage is drastically reduced by requiring the last-level cache to track only the clusters—not the individual cores inside each cluster. Global traffic is also reduced since portions of coherence transactions are handled inside the clusters, thus eliminating inter-cluster communication. As a direct result of intra-cluster locality, the last-level cache sends a single *invalidation* message to a cluster and receives a single *acknowledgment* message from that cluster to invalidate a data block from all caches inside the cluster.

There are already commercial systems that benefit from clustered architectures, such as NVIDIA's Fermi GPU [33], Sun/Oracle's T2 [38], and AMD's Bulldozer [8]. Furthermore, proposals such as Rigel [21] are based upon clustered architectures.

Motivation: But despite the arguments in support of clustered cache hierarchies, there are also obstacles to overcome as a prerequisite for their wide adoption by industry. The prevalent obstacle is the complexity and cost due to the coherence that must be implemented. For example, a *hierarchical*, invalidation-based, MOESI directory protocol in GEMS has a very high number of states, mainly in the intermediate-levels of the hierarchy. Why is that?

Fundamentally, invalidation-based, directory coherence must perform two functions:

1) Invalidation upon write: upon a write miss, invalidate all other sharers
2) Indirection and downgrade: upon a read miss, find the latest written value and downgrade the writer

These two functions enforce the Single Writer Multiple Reader invariant and ensure that written values are propagated correctly [39]. The complexity of a flat (non-hierarchical) directory providing this functionality is well understood and although there is ample implementation experience, there are also significant advantages in simplifying even this case [5], [22], [13], [10], [9], [37], [35], [16], [19]. In the case of a hierarchical clustered cache architecture, directory-based coherence becomes significantly more complex: it must also be performed hierarchically. A clustered cache hierarchy is handicapped if coherence is not implemented using a hierarchical directory and a hierarchical (tree) protocol [1], [26], [34], [25], [18], [30], [31], [11], [23], [12], [31], [32]. A single flat directory at the root of the hierarchy (e.g., the LLC) simply negates the scalability of the whole approach and proves problematic in handling caching in intermediate levels between the root (LLC) and the leaves (L1s).

Thus, both the invalidation and the indirection/downgrade functions have to be performed hierarchically. This means that intermediate nodes must have the ability to simultaneously behave both as root caches/directories (i.e., send invalidations, collect acknowledgements, indirect requests, as does the LLC) *and* as leaf caches (i.e., respond to invalidations and/or downgrades, as do the L1s). Moreover, one personality (leaf or root) can invoke the other recursively. For example, invalidations

treat nodes in intermediate levels both as leaf nodes to be invalidated but also cause them to behave as root nodes initiating new invalidations in their sub-cluster (similarly for downgrade requests). It is this dual behavior and the resulting cross-product of the states of the two personalities (root and leaf) in intermediate levels that increases the implementation complexity to prohibitive levels. Verification becomes inordinately costly and time to market may be dangerously compromised.

Approach: Our approach in addressing this problem is to simplify the source of the complexity: invalidations and downgrades. To the best of our knowledge this is the first paper that studies such simplification in the context of clustered cache hierarchies. Inspired by recent work, we provide a new solution to hierarchical coherence based on the principles laid down by the VIPS [37] and DeNovo work [9]. Based on data-race-free (DRF) semantics [2], we eliminate invalidations with self-invalidation on synchronization points. We eliminate indirection and downgrades by using a write-through policy (also referred to as self-downgrade) [37]. However, self-invalidation and self-downgrade in a clustered hierarchy, if not done intelligently, can decimate caching in the intermediate levels, effectively rendering the clustered hierarchy into a flat two-level L1/LLC hierarchy.

Contribution: Our main contribution is to define *hierarchical, selective* self-invalidation and self-downgrade for clustered caches. In previous work, classification of data into private and shared, proved invaluable for selectively applying self-invalidation and self-downgrade: only data classified as shared are self-invalidated in the L1s and follow a write-through policy—private data are excluded [37]. The key advancement in our work is to redefine the notion of private and shared in a clustered hierarchy. We make the following observation: data can be shared entirely within one cluster but can be *private* to this cluster when viewed from the outside. In a multilevel hierarchy, this notion is applied recursively starting with the levels closest to the cores.

Example: Assume now that we have a hierarchical private/shared classification that can identify the *level* in which each data block is shared, i.e., the *common shared level* that captures all sharing for a block. Figure 1 shows an example. We use a hypothetical 8-core system with 4 cache levels (L1–L4) and a degree of 2 (i.e., binary tree). The system can be subdivided into many clusters at various levels (L1–L4). One cluster at each level is shown. In this example, data block D is private to core 7 (L1 cluster). Data block A is shared between cores 0 and 1. It is, however, *private* to the L2 cluster that contains these two cores. Similarly, block C is shared by cores 5 and 6 inside the L3 cluster containing these cores. Finally, block B, shared by cores 2 and 4, is shared at the L4 (system) level. We perform this classification dynamically as data blocks are accessed by cores. For efficiency, we use the page table to detect the level of sharing at page granularity. A page is classified as shared at the highest-level cluster that encompasses the sharing of all its blocks. Once the classification is performed for a page, self-invalidation and write-through of all its blocks become localized to the cluster wherein it is shared. For example, in Figure 1 block A is self-invalidated in the L1s, but not outside the L2 cluster (its private chain of copies in the L2, L3, and L4 is left undisturbed). Block A is also written-through to the L2, but no further—its private copies outside the L2-cluster follow a write-back policy. Similarly, block B is self-invalidated in all the levels between L1 and L4 (exclusive), and it is written through to the L4.

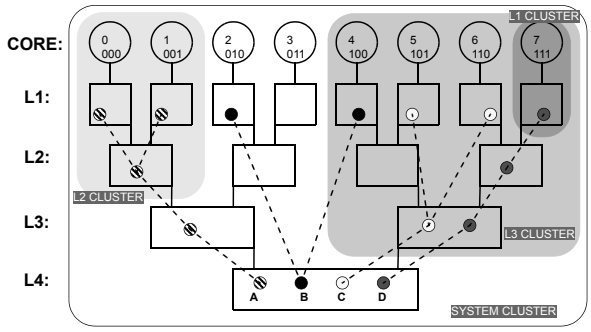


Fig. 1. Hierarchical classification in a clustered hierarchical cache architecture

Paper Structure: To present our approach we start with hierarchical private/shared classification (Section III) and subsequently explain how the self-invalidation and self-downgrade of a data block are restricted to operate in the sub-cluster where this block is shared (Section IV). We examine two alternatives on how to use the intermediate levels of caches between the common shared level of a data block and the L1s. The first option is not to store a shared block in intermediate levels and the second is to store *clean* copies (updated with cascading write-throughs). These options affect whether self-invalidation and write-through are applied only to the L1s and the common shared level respectively, or they cascade the cluster. Finally, we discuss the relation of our approach to the recently proposed Heterogeneous-Race-Free models, HRF-direct and HRF-indirect [17] (Section V).

Results: We evaluate our approach by simulating 16-core and 64-core hierarchical topologies using GEMS and a collection of all the SPLASH-2, PARSEC, and other benchmarks (Section VI). We compare against the hierarchical directory MOESI protocol and the TokenCMP protocol provided in the GEMS simulator and the VIPS-M protocol [37]. Our results show comparable or better performance and reduced network traffic compared to both MOESI (12% less traffic than MOESI in 64 cores, with significant reductions in 19 out of 22 benchmarks) and VIPS-M, which in a hierarchical configuration yields similar figures as MOESI. Compared to Token our protocol obtains similar performance but better scalability, especially in network traffic.

II. Background

A. Hierarchical coherence

Extending directory-based coherence protocols from flat to hierarchical architectures has been extensively studied. Hierarchical implementations allow more scalable CMPs by mitigating two overheads of flat directories, storage cost, and global traffic, by exploiting intra-cluster locality. However, the intrinsic complexities of directory-based coherence not only remain unresolved but are exasperated.

Wilson Jr. [1] shows the benefits of a hierarchical multiprocessor architecture by extending the shared snoopy buses to multiple hierarchies. He further introduces a clustered design in which the memory is distributed among clusters. Maa *et al.* [26] introduce distributed directories—tree directory and hierarchical full-map directory—to deal with the storage overhead of large-scale multiprocessors. Their hierarchical full-map directory drastically reduces the storage overhead by exploiting the sparsity which enables them to have set-associative intermediate directories. Their proposal averages performance gain over a flat full-map directory by exploiting locality. Locality allows majority of the transactions to be performed within a cluster without having to traverse the whole hierarchy. The hierarchical directory further benefits from shorter access times due to having small intra-cluster directories. Nilsson *et al.* introduce their tree-based directory coherence protocol, known as Scalable Tree Protocol (STP) [34] to overcome the performance bottleneck associated with SCI [14] due to write latency. STP dynamically arranges caches in an optimal tree structure and adopts an efficient replacement policy to keep the tree structure balanced and optimal upon replacements. Similar to SCI, STP uses pointers to form the tree. Although its implementation cost is slightly higher than SCI, STP incurs lower write latency which reduces the overall execution time. STP can be seen as a performance-optimized version of SCI.

Lenoski *et al.* [25] introduce *Dash* architecture to overcome the bottleneck problem of centralized directories. Their solution relies on: i) partitioning and distributing the directory and main memory among clusters, and ii) a coherence protocol optimized for such a distributed directory architecture. Distributed memory allows the system to exploit locality, resulting in reduced traffic and latency. To keep the hierarchy coherent, *Dash* employs a mixed coherence policy: intra-cluster snooping buses and inter-cluster directory-based coherence. While the *Dash* architecture is more scalable than a flat one due to intra-cluster locality and the reduced overhead of a centralized directory, having a mixed coherence policy adds complexity, especially when verification is concerned.

While previous efforts perform independent optimizations for the coherence protocol and the interconnection network, Kaxiras and Goodman propose the *GLOW* extensions [18] for SCI, to enable concurrency for *reads* and *writes* (invalidations), in the network. They implement the coherence protocol at the network switches and dynamically create sharing trees on arbitrary topologies based on *geographical* (topological) locality. Subsequently, Eisley *et al.* introduce *In-Network Cache Coherence* [11] by integrating coherence layer in the interconnection network fabric. They remove directories from the nodes and integrate in each router in the network *virtual tree caches*, which form virtual links to track sharers of data. The resulting coupled coherence and interconnection network enables in-transit optimizations, such as reducing the latency of read/write coherence transactions by minimizing a series of end-to-end messages—between requestor, directory, and sharers—needed to satisfy a coherence transaction to a single round trip between the requestor and responder nodes. This results in latency reduction since invalidating the sharers is done in parallel with routing the *write* request towards the *home* node and invalidation acknowledgments arrive at the *home* node shortly after the *write* request.

To address the serialization bottlenecks and also non-determinism in hierarchical shared caches imposed by using timeouts, Ladan-Mozes *et al.* introduce Hierarchical Cache Consistency (HCC) [23] which embeds the coherence protocol in the interconnection network, resulting in distributed scalable CMPs. They ensure progression and deadlock freedom in the protocol by requiring a *unique path* between each core and each memory bank. However, HCC inherits complexities of MSI-/directory-based protocols, resulting in a transition table for shared data with 93 entries.

While migrating from flat to hierarchical architectures results in higher scalability and performance, formal verification of the resulting coherence protocol becomes by orders of magnitude more difficult than that of the flat version. There are works that put verification first and design hierarchical/clustered coherence protocols from the ground up, engineered for formal verification. Marty *et al.* separate *correctness substrate* from *performance policy* by introducing *token coherence* for multiple-CMP systems [30]. The resulting TokenCMP protocol exhibits flat behavior from correctness standpoint—easily verifiable— and has hierarchical characteristics. Moreover, Zhang *et al.* introduce *fractal coherence* [44]. By designing the coherence protocol in a fractal fashion, the verification of the whole system is limited to the verification of the base cluster, which is small and manageable. The only requirement is to show that the whole cluster exhibits fractal behavior, which is within the budget of the available equivalence-checking tools.

How we differ: We take a different approach to hierarchical, clustered, coherence. Instead of trying to alleviate the limitations of directory-based invalidation coherence and of its verification, we ask the question: can very simple coherence mechanisms (e.g., strictly request–response with no indirection) such as self-invalidation and self-downgrade, be applied to a hierarchy? And how do we make them efficient? Our aim is to burden a single uncommon function, private/shared classification, with the complexity of the hierarchy and delegate this to software but relieve other common functions (implemented in hardware) from hierarchical complexity. This is the philosophy pioneered in Dir_1-SW [42].

B. Simplifying coherence

Recently, a number of proposals aim to simplify coherence by relying on data-race-free semantics and on self-invalidation to eliminate invalidation traffic and the need to track readers at the directory. With the addition of self-downgrade, the directory can be eliminated [37] and virtual cache coherence becomes feasible at low cost, without reverse translation [20]. The motivation for simplifying coherence has been established by many [24], [19], [9], [40], [6], [37], [20] and we will not add significantly by reiterating here. Suffice to say that significant savings in area and energy consumption without sacrificing performance have been demonstrated in many recent papers [19], [9], [40], [6], [37], [20]. Additional benefits regarding ease-of-verification, scalability, time-to-market, etc., ensue as a result of simplifying rather than complicating such fundamental architectural constructs as coherence.

Our work takes the approach of simplicity of these efforts, but now to the domain of hierarchical clustered cache

architectures—which, to the best of our knowledge, has not been attempted before.

III. Hierarchical Data Classification in Clustered Hierarchies

Previous proposals (e.g., VIPS-M [37]) perform private/shared data classification [15], [22], [10], [35], [16], [5], [43] to optimize coherence in a *flat*—two-level—multi-core system. Even if there are more than one level of private caches (e.g., private L1–L2 and shared L3) the classification remains the same. In VIPS-M, the private/shared data classification is used to filter self-invalidation and write-through in the L1s.

In a hierarchical cache architecture the private/shared data classification takes a new dimension. Now, we are not interested in knowing only if data are globally private or globally shared but also in knowing where they are shared, i.e., if the data are shared across the whole system or only inside a particular cluster. *This classification restricts the application to perform self-invalidation and write-through within a specific cluster. Our aim is to encapsulate the complexity of hierarchy to just this uncommon operation—determining cluster sharing—while simplifying all other common operations (reads and writes).*

Before we continue, we give some nomenclature we use in this paper. In a hierarchical cache architecture, the level of a cache corresponds to the conventional naming of caches, e.g., the level of an L1 is 1 and is the *lowest* level, and the level of an L4 is 4. If, for example, L4 is the LLC then it is the *highest* level and the *root* of the hierarchy. In any sub-tree, also called *cluster*, the cache at the highest level of this sub-tree is its *root* cache. Leaf caches are always the L1s. Any cache between the leaves and a root is an *intermediate* cache. For the rest of the paper we discuss symmetrical, constant-degree, fully populated hierarchies, but one can easily extend our algorithms to any other partially-populated, or non-constant degree, or asymmetrical hierarchy.

A block can be shared entirely within a cluster and not outside. For example, if the block is in just two L1s which share the same L2 in a small cluster, then the block is known as *shared* in the L1s. But from the outside, the block is known as *private to the cluster*. In the leaf and intermediate caches, we only need to know that the block is shared (self-invalidates and follows a write-through policy). Outside the cluster, we need to know the *level* where the block changes from private to shared; in other words, the level of the *root cache* of the cluster. We call this the *common shared level* (*CSL*) for this block. In the example above, the shared block between the two L1s is *private* in L3 and L4 seen from L2 (the block's CSL is 2). The actual L2 that has this block *privately* needs to be known for various operations. However, its identity can be derived by knowing only the first core that accessed the block and the block's CSL. We show this below.

If a new core requests a shared block from outside the cluster where a block is shared, then the block's sharing level must change. We use the page table to detect changes in the sharing level at a page granularity. We do this to minimize the number of transitions since: i) the sharing level of a whole page—not each individual block—changes at once; and ii) page-level transitions can happen only when a core first accesses a block and thus has to take a TLB miss. In contrast, classification at block granularity would entail transitions for each individual block on cache misses, which are far more numerous than TLB misses. While at page level we have a coarser grain, less accurate classification (leading to more blocks classified as shared at higher levels—i.e., more globally), the transitions are far fewer and therefore their cost is not as critical. The examination of the trade-offs when classification is performed entirely at block granularity is left for future work. Here, sharing at page granularity defines the CSL of all the blocks in a page. This approach inherits the same well-known traits of other page-level classification approaches and has to be applied accordingly.

A. Detecting sharing-level changes

One of the first problems to solve in a hierarchical clustered cache architecture is how to detect the common sharing level and its changes. There are several potential algorithms that give this answer. We are interested in an implementation that stores as little information as possible and updates this information the least amount possible.

Since we are performing hierarchical classification at a page granularity, CSL changes are detected on TLB misses. Associated with each page table entry is the core that first accessed this page and the current common sharing level. The first core that accesses a page is the only owner of the page (globally private) and CSL is set to 1. If another core attempts to access the same page then a new CSL is derived by comparing the ID of requesting core with the ID of the original owner. Assume that core IDs are *n*-bit numbers. For a hierarchy of a degree of d we divide the core IDs into segments of $log_2(d)$ bits. We compare pairwise segments of the two IDs starting from *most* significant end. The position of the first pair of segments that differ, identifies the CSL: $CLS\_level = segment\_position + 1$

Example: We show an example, in Figure 2, with 8 cores and a binary hierarchy (e.g., 1 bit segments). The first core that accesses the page is 001. If the second core is 000 (Figure 2, top example), then they differ in the first segment (indicated with black background), ergo the CLS changes to 2. The L2 that is shared at this level is always identified by the most significant segments that are the same: 00 (indicated with blue background). If on the other hand, the second core is 011 (Figure 2, middle example), it differs from 001 in the second segment; therefore the CSL changes to 3. The shared L3 cache is still identified by the most significant segments that remain the same: 0. Similarly, if the second core is 111 (Figure 2, bottom example), it differs from 001 in the most significant *third* position, thus the CLS changes to 4. The shared cache is L4 which is uniquely identified. If a requesting core differs from the first core in a position that gives a CSL that is less than the current CSL, then the requesting core is already in a sub-tree where the data are identified as shared. This algorithm works because the first core that accesses a page defines how the sharing sub-tree will grow. The advantage of this algorithm is that a core ID is only stored once per page (for the first core) and never needs updating. The CSL for the page, however, may be changed as new cores are requesting the page.

Re-classification and thread migration: Currently re-classification from shared to private is only performed upon

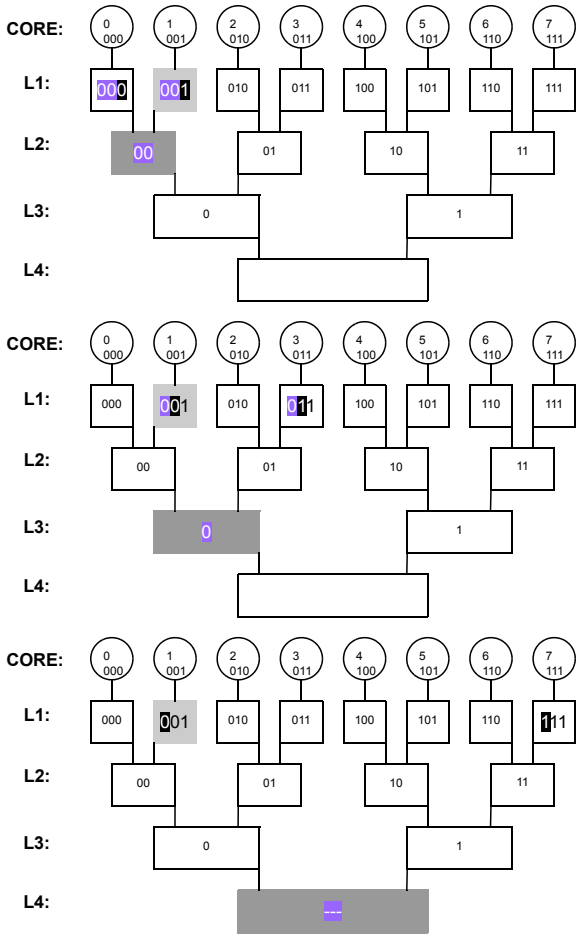


Fig. 2. Detecting the Common Shared Level (CSL)

page evictions from main memory. Since classification is done in software it can be changed to an adaptive approach, using for example decay techniques [36]. This is future work. There is, however, a case were we do perform re-classification and this is for private data on thread migration. As in previous work [15], private pages are self-invalidated and shot down from the TLB of the last owner core in the event of migration and the page owner in the page table is changed to the destination core as the TLB entries are reloaded. Thread migration does not affect already shared pages unless threads are migrated to new clusters. This might increase the CSL of the pages and—in the absence of re-classification—can be taken under consideration in the migration algorithm.

B. Encoding the classification

Once we detect a change in the CSL the question is where do we encode this and how do we use it. The current CSL of a page and its first owner are always associated with the page table entry (PTE). We preferably save these within the PTE if there are available unused bits, or alternatively, in a separate memory structure. We assume that this information will be cached in the system (last-level) TLB, if one is available. The overhead is low since we only need $log_2(N)$ bits for the first owner and $\lceil \log_2 \lceil \log_2 N / \log_2 d \rceil \rceil$ for the CSL, in a system with N cores and a hierarchy of degree d.

However, we also need per-page CSL information to be readily available to restrict self-invalidation and write-throughs to the appropriate cluster, independently for each page. Fundamentally, there are three operations in our approach:

- **Self-invalidation (SI)**: Self-invalidates data of a page in all the leaf and intermediate caches up to (but excluding) the CSL. From the CSL onwards (i.e., to higher levels) the page is considered private and does not self-invalidate.
- **Self-downgrade (SD)**: We propagate write-throughs for this page from the L1s all the way to the CSL but not further.
- **Recovery**: Finally, when the CSL changes we need to propagate all the modified data that reside in the old CSL cache to the new CSL cache and globally update the CSL information. Essentially this is the only example of a *forced* downgrade, similarly to other protocols, but we restrict it to classification where it is uncommon. To distinguish it from much more common self-downgrade, we call this operation *recovery*. Recovery also exists in VIPS-M: when a page transitions from private to shared, the private L1 *owner* must be notified so it can make the modified data of this page visible to the LLC and start self-invalidating them at synchronization [37].

In our approach, we store only the CSL—no owner field—in the core TLB entries. Cache lines do not need to store CSL info, just a private/shared (P/S) bit. This has the advantage of the CSL being available a-priori, at the time when a request is generated, allowing for the possibility of skipping intermediate cache levels and going directly to the CSL cache. This ability is useful when intermediate caches do not store shared data. Knowing the CSL would allow us to write-through directly to the CSL cache and optimize atomic operations which only concern the CSL cache and not any intermediate cache.

Recovery: Recovery of a page (increasing its sharing level) concerns all TLBs that contain an entry for this page. We must ensure that the correct (new) CSL information is communicated to all the cores that can have a copy of the PTE in their TLB because we need to change the level of the future requests for this page. Potentially this includes *all* the cores of the cluster whose root cache is the old CSL cache.

To *recover* a whole sub-cluster, we first need to ensure that all the TLBs in cluster are locked.[1] This is achieved with core-to-core interrupts (a feature which is available in many architectures). The core causing the change in the CSL interrupts all cores whose root cache is the old CSL. We wait until there are no MSHR entries for the page—no pending requests for the page. Then all cores block any new requests for that page (lock bit in the TLB) and send a forward recovery to the shared cache.

Subsequently, we must *self-downgrade* all the *dirty* blocks of the page being recovered, from the old CSL cache to the new one, and change the policy of all blocks (in the old CSL cache) from write-back to write-through by setting their P/S

[1]We relax this when a TLB does not hold the corresponding PTE, since the page table entry itself is locked by the core causing the recovery.

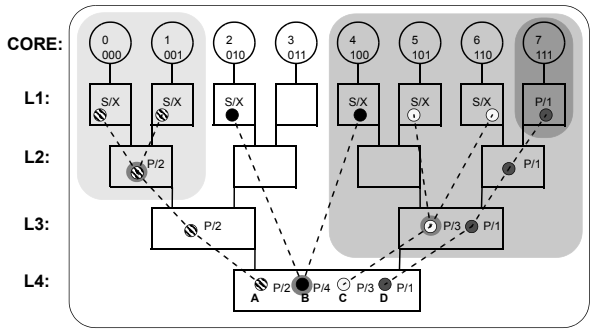


Fig. 3. Two-Level policy for shared blocks. Intermediate levels between the L1 and the CSL of a shared block do not cache this block. There are no restrictions from the CSL to the LLC for the chain of private copies (indicated by "P/CSL").

bit to S. If the recovery is only one level up, the only cache to recover is the old shared level cache. However, if the recovery is n levels up, we must recover all the caches of the next $n-1$ levels towards the new CSL. This is because all the dirty data present in any intermediate cache must be reflected in the new CSL cache.

When the recovery of the old CSL cache is done, acknowledgements are sent to the TLBs that are locked. The acknowledgment updates the CSL of the TLB entry and resets the lock bit in the TLB. One of the cores (e.g., the core with smallest ID in the cluster) unlocks the page table.

Discussion: Recovery of a page is an expensive operation. However, it is offset by the fact that it is quite rare. It only happens a few times per page (no more than $\#HierarchyLevels - 1$ per page). For this reason, it is the operation of choice to burden with the complexity of a hierarchy, allowing for much more common operations (reads, writes, self-invalidations, and self-downgrades) to be implemented more efficiently. Furthermore, we choose to support this operation in software. Software can be changed, debugged, and verified using program verification techniques. Thus, CSL management and coherence operations are separated so that the protocol components can be verified with a divide-and-conquer approach.

C. Read-only classification

A different type of classification that is especially useful with self-invalidation is *read-only* (*RO*) classification. Shared read-only data can be excluded from self-invalidation [37]. Read-only classification can be easily implemented, relying on the RO bits of the PTEs. When a page transitions from RO to read-write (RW), we must also perform a recovery to notify all the cores that share this page about the change, so they can start self-invalidating the corresponding cache lines.

The classification change from RO to RW inside the cores can be deferred until the next synchronization in each core. The only requirement is that all the cores in the cluster where the page is shared be notified about the RO-to-RW change before the core that causes the change passes a synchronization. This is done by sending notifications and collecting acknowledgements, without the need to block any cores. The critical observation is that a RO-to-RW transition imposes no cost, expect in the uncommon case where a core that caused such transition may have to wait for the acknowledgements to pass its next synchronization.

IV. VIPS-H: A Hierarchical VIPS

Given one of the hierarchical data classifications described in the previous section, here we explain the behavior of a coherence protocol relying on self-invalidation and self-downgrade of shared data blocks.

The main policy decision that affects the implementation of these operations concerns how we use the intermediate caches between the root cache and CSL and between the CSL and the leaf caches *per data block*:

Intermediate levels between the root cache (LLC) and a CSL cache. A block is private between the LLC and its CSL (P/S bit set to P). For performance reasons we allow copies of the block to exist between the LLC and the block's CSL, but we do not enforce inclusion. When satisfying a request at a level higher than the CSL (because the CSL and possibly other levels have evicted) we rebuild the private chain from the LLC to the CSL by copying the block in all the levels where it is missing. The P/S bits of these copies are set to P (and the correct CSL is copied in them, if we encode it in the caches).

Intermediate levels between the CSL cache and the leaf caches (L1s). A block is shared between its CSL and the L1s (P/S bit set to S). We discern two policies:

- Strictly two-level policy. A shared block exists only in the L1s and in the CSL, but not in intermediate caches. The advantage of this approach is the simplicity in self-invalidation (which is strictly restricted in the L1s) and potentially in self-downgrade (if the CSL is known and intermediate levels can be skipped). Additionally, read misses can skip the lookup in intermediate levels. The recovery operation must self-invalidate the old CSL, in addition to self-downgrade. The disadvantage is lower performance from more costly misses. This policy is shown in Figure 3 where all sharing is strictly two-level.
- Multilevel policy. A shared block can exist in any intermediate level between the L1s and the CSL. The advantage is higher performance, but self-invalidation and self-downgrade must now *cascade* all the levels between the L1 and the CSL. This policy is shown in Figure 4 where sharing is multilevel.

In the sections below, we describe the protocol design starting with the more frequent events (read and write misses), to less frequent events (atomic operations, self-invalidations, and self-downgrades).

DRF memory accesses (Loads and Stores): A DRF miss searches for the block in the cache hierarchy, starting from the first level and stopping at the level where the data are found. The response copies the data in intermediate caches. For every copy we set its P/S bit that indicates whether it is invalidated on self-invalidation. To set the P/S bit we need to know the CSL, which comes from the TLB. For levels below the CSL (i.e., between the L1 and the CSL) we simply set the P/S bit

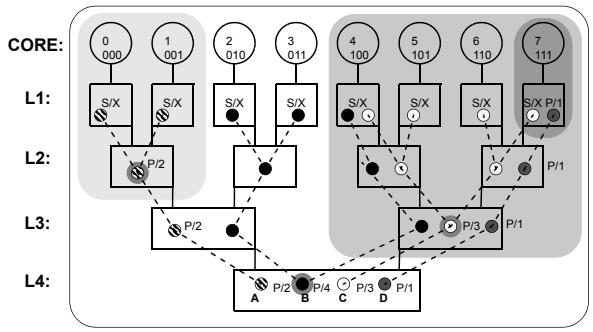


Fig. 4. Multilevel Policy: A shared block can be cached in any intermediate level between the L1 and its CSL. There are no restrictions from the CSL to the LLC for the chain of private copies (indicated by "P/CSL").

to S. When we follow a strictly two-level policy we skip the intermediate levels between the CSL and the L1.

A DRF store writes in the L1 and is always a hit. No invalidations and no extra latency is incurred. Out of the critical path of the store, the data block is requested (as in a load request) and when it arrives it is merged with the modified words. When the L1 cache line is self-downgraded, the write through of the dirty words (i.e., the diff of the cache line as in [37]) *cascades* and updates all the shared copies of the intermediate levels until it finds a private copy (at the CSL or greater level). At that point the write-through stops and merges the diff in the data block. Levels that have evicted the copy are simply skipped.

The invariant of our approach is that we do not allow dirty blocks in intermediate levels. We only store dirty blocks in the L1 cache and in the CSL or higher levels. This means that we do not need dirty bits per word at any cache level (only in the L1 MSHRs to create diffs as in [37]). Dirty data in the CSL or higher levels, use a *write-back* policy (since they are private) and only a single dirty bit per cache line is needed.

Evictions: Evictions of clean lines are silent. An eviction of a dirty line can cause a write-through or a write-back depending on where it is in the hierarchy. Since we only allow clean copies in the intermediate levels between the L1 and the CSL an eviction can cause a write-through only in the L1 (where we have create diffs). Write-throughs cascade to the CSL or higher level, updating all the intermediate caches that have the block. With the strictly two-level policy, intermediate caches are not updated. Write-backs simply write the whole block into the next cache level.

Non-DRF memory accesses (Atomics and other): Atomic, read-modify-write requests always operate at their CSL and no other level. Since the CSL is known from the TLB, all Intermediate levels can be skipped as an optimization. The hierarchy is searched for a private line. If this line is not at the CSL but higher (towards the LLC), it is copied in all the levels, from where it is found all the way to the CSL. At this point the atomic request has reached the CSL and blocks the requested line. When the atomic is resolved, it either writes or sends an unblock message to the CSL, so other atomics can proceed. Our approach does support arbitrary data races as long as they are intended and identified. Using the proper fences (see below) racing accesses can be implemented in any self-invalidation/self-downgrade protocols. In these cases, competing accesses meet directly in the CSL.

Self-invalidation and self-downgrade fences: In SC for DRF, synchronization is exposed to the hardware [2]. We consider that *fences* in the program perform this job. A *release* operation corresponds to a *self-downgrade fence* (*SD* fence) that completes all outstanding write-throughs. An *acquire* operation corresponds to a *self-invalidation fence* (*SI* fence) that causes the self invalidation of shared data. We have annotated the synchronization points of all the benchmarks used in the evaluation accordingly. In a hierarchical clustered architecture these fences operate as follows:

- SI fence: In the strictly two-level policy the SI fence operates exclusively in the L1s as in VIPS-M [37]. However, in the multilevel policy the self-invalidation cascades to all cache levels from the L1 to the LLC. At every level it performs a 1-cycle flush by bulk-resetting the valid bits of the shared (non-read-only) lines. Self-invalidation flushes all the blocks whose CSL is higher than the level they reside. This is guaranteed by the way their P/S bits were set.
- SD fence: SD concerns the first level. Cache line diffs are written-through as explained above (DRF memory accesses). The SD fence awaits for the completion of the write-throughs of all the lines that are temporally dirty and have an allocated MSHR.

V. Heterogeneous-Race-Free (HRF) Memory Models and HRF-Dynamic

Coherence such as VIPS [37], DeNovo [9], SARC [19], and of course our proposal, are intimately connected to the synchronization model since they rely on data-race-free (DRF) semantics and synchronization exposed to the hardware to deliver SC for DRF [2]. But in a hierarchical clustered cache architecture, what is DRF? Hower *et al.* pose this question in their work on heterogeneous-race-free models [17]. The issue at hand is scoped synchronization which operates locally within a cluster vs. globally-scoped synchronization which operates across clusters. The authors propose two memory models: HRF-direct and HRF-indirect [17].

- HRF-direct: transitivity is only guaranteed for same-scoped synchronization.
- HRF-indirect: transitivity is guaranteed for different-scoped synchronization.

How HRF models relate to our approach? Taking the analogy from GPUs to a general clustered architecture the scope of any synchronization depends on the CSL of the synchronization variable. Assume now that we have scoped synchronization (e.g., it is offered in the programming model). In order for cores in a cluster to synchronize, the synchronization variable (lock, barrier flag, etc) must be shared at the level of the cluster's root cache. Accesses inside the cluster separated by this synchronization are DRF. However, if any core in the cluster synchronizes with a core in a different cluster, a new CSL (which encompasses both clusters) is established for the synchronization variable. If the new cluster attempts to access data that were private in the first cluster then

such data become (through the process of recovery) shared in the new CSL, before the access is allowed to proceed. Thus, the latest values of the data are exposed (on demand) to the new cluster. This guarantees the transitive behavior dictated in the HRF-indirect model. It follows that our approach provides SC for DRF in the presence of scoped synchronization.

However, the interesting property of our approach is that it also provides the *benefits* of HRF-indirect even if one does not assume scoped synchronization. Viewing it from a different perspective, even if one imposes an HRF-direct model and all synchronizations must be non-scoped, or globally-scoped, to provide DRF guarantees across all cores, the benefits of scoped synchronization are obtained dynamically. Globally-scoped synchronization does not necessarily mean global sharing. This is due to the fact that the common shared level of race-free data is dynamically set at least as high as the highest level of any synchronization variable used to synchronize conflicting accesses between any two cores (even transitively). If the synchronization is confined within a cluster, the sharing is generally (but not always) confined within the same cluster.

Because of such dynamic behavior and since our approach delivers the benefits of scoped synchronization dynamically (even in absence of scoped synchronization as such), we venture to suggest the possibility of other models, such as for example an *HRF-dynamic* model that bridges the performance gap between HRF-direct and HRF-indirect with dynamic optimizations. We do not provide a formal definition for HRF-dynamic since this is not the focus of our paper, but point out that the discussion started by Hower *et al.* [17] may lead to new models encompassing dynamic behavior and optimizations.

VI. Evaluation

A. Simulation environment

In our evaluation we use Wisconsin GEMS [29], a detailed simulator for multiprocessor systems. We model in-order cores that along with the Ruby cycle-accurate memory simulator (provided by GEMS) offers a detailed timing model. The interconnect is modeled with the GARNET network simulator [3]. We model hierarchical systems comprising three cache levels: the L1 is private to each core, the L2 is partially shared, i.e., shared only within a cluster, and the L3 is globally shared. The three level limitation is imposed by the implementation of our base case. We simulate both 16-core and 64-core systems. The 16-core system has four clusters of four cores each (4×4). The 64-core system has four clusters of 16 cores each (16×4). All L1s are connected to the L2 cache in the cluster and all L2s are connected to the L3, thus creating a hierarchical network. More details about the configurations are shown in Table I.

We evaluate the proposed VIPS-H protocol with two caching policies (two-level and multilevel), and we compare them with the directory-based hierarchical MOESI and hierarchical token-based (TokenCMP [30]) protocols provided with GEMS (from now on, H-MOESI and H-Token), and the VIPS-M protocol [37]. We note that H-MOESI employs *unlimited* directory caches and is hardwired for only three cache levels (hence the limitation in our comparisons). Changing the implementation to realistic limited directory caches, would increase even more the number of states of the protocol and would worsen performance. We chose H-Token as it has a similarly low complexity as VIPS-H. H-Token uses local (intra-cluster) and global (inter-cluster) broadcast, without further filtering optimizations [30]. More sophisticated versions of Token, e.g., Token-M [28], could potentially perform better but at a penalty of additional complexity and cost and therefore not considered. VIPS-M classifies pages either as private or as globally shared and do not store blocks belonging to shared pages in intermediate levels, so the selective self-invalidation only need to be performed at the L1 caches as in [37].

TABLE I. System configuration

Parameter	Values
Processor frequency	3.0GHz
Block size / Page size	64 bytes / 4KB (64 blocks)
MSHR size / Delay timeout	16 entries / 1000 cycles
Split L1 I & D caches	32KB, 4-way, hit time 1 (tag) + 1 (data) cycles
L2 cache	4MB, 16-way, hit time 6 (tag) + 6 (data) cycles
L3 cache	16MB, 32-way, hit time 10 (tag) + 20 (data) cycles
Memory access time	160 cycles
Flit size	16 bytes
Message size	72 bytes (5 flits) data, 8 bytes (1 flit) control
Switch-to-switch time	6 (on-chip), 13 (off-chip) cycles

We employ a wide variety (22) of parallel applications in our evaluation. In particular, we simulate the *entire* Splash-2 suite [41] with the *recommended* input parameters. We also run six benchmarks from the PARSEC benchmark suite [7], all of them with the *simmedium* input, except *Streamcluster* and *Swaptions* that use the *simsmall* input due to simulation constraints. Finally, we simulate two additional applications: *Em3d* (38400 nodes, 15% remote) is a shared-memory implementation of the Split-C benchmark. *Tomcatv* (256 points, 5 time steps) is a shared-memory implementation of the SPEC benchmark. We simulate the entire application, but collect statistics only from start to completion of their parallel section. Our results account for the variability in multithreaded workloads, showing the corresponding error bars [4].

B. Complexity, cost, and area

One of the main characteristics of VIPS-H is its low complexity. As a metric to show the complexity of VIPS-H compared to the other two hierarchical approaches analyzed in this work: H-MOESI and H-Token, Table II shows the number of total states and base states required for the implementation of each protocol and each cache controller. As we mention previously, intermediate levels in H-MOESI are the cause of the enormous complexity requiring the disproportionate number of 59 protocol states, 13 of which are base states. On the other hand, both VIPS-H and H-Token have most of its complexity in the L1 controllers. However, the L1 of VIPS-H is even more simple than the L1 in the other protocols since L1s initiate the transactions to other levels but other levels never issue requests to the L1s —only responses). The base states in VIPS-H correspond to the Invalid, Clean and Dirty states.

Regarding area requirements, VIPS-H does not use any directory structure, which significantly reduces the area overhead entailed by coherence management and increases scalability by removing congestion points. VIPS-H includes one P/S bit per cache line, whose only function is to filter self-invalidation and self-downgrade transactions. On the other hand, H-Token

TABLE II. STATES AND STORAGE COST FOR A 64-CORE 16×4 SYSTEM

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

TABLE III. EVENTS, FREQUENCY, COMPLEXITY AND EFFICIENCY

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

requires a counter of tokens, (plus one owner token) per cache line ($1+\lceil log_2 cores \rceil$). Finally, H-MOESI requires full bitmaps with as many bits as cores in a cluster in the L2 directory and as many bits as clusters in the L3 directory.

Assuming a commonly employed in-cache directory, the storage required by H-MOESI for the coherence information is 844KB, as show in Table II (recall, however, that we evaluate H-MOESI with *unlimited* directories). For this number we account for the bits required to code the states ($log_2 b$), where b is the number of base states, and the coherence information. The storage required by H-Token is 584KB (69% of H-MOESI). The storage required by VIPS-H is only 204KB (24% of H-MOESI). Regarding scalability, it is important to note that the storage required by VIPS-H grows in constant order, except for the TLB's CSL field that grows as $\lceil \log_2 \lceil \log_2 cores / \log_2 degree \rceil \rceil$.

But how can VIPS-H, having so low complexity, achieve performance numbers similar to H-MOESI? The answer lies in Table III, which shows frequencies (in percentage of events per instruction), complexity, and efficiency of the possible events in VIPS-H. The most frequent events are resolved in a simple way and are very efficient, while the less frequent events carry all the complexity and inefficiency, requiring cache blocking or software intervention.

C. VIPS-H hierarchical behavior

Figure 5 shows the fraction of pages that, at the end of the application, have their CSL in the L1 (i.e., private), L2 (i.e., partially shared), and L3 (i.e., globally shared). As expected there are few pages in L2, since applications typically have either private data or highly shared data. On average, the CSL of about half the pages is the L3 for a 16×4 configuration, although the trend when increasing the number of cores favors private data. The large number of L3-CSL pages is also due to the lack of clustered locality even when we have low degrees of sharing. We run all our benchmarks unmodified and have not attempted to affect thread placement or data sharing to maximize the benefits of clustering. Such orthogonal optimizations on the software side can help hierarchical coherence to perform and scale better.

Figure 6 shows for every L1 miss the target CSL (first bar) and the eventual level where they are resolved in the two-level

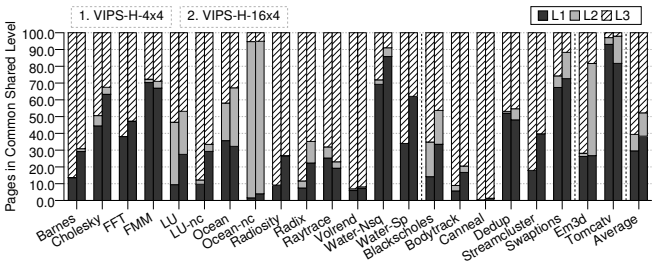
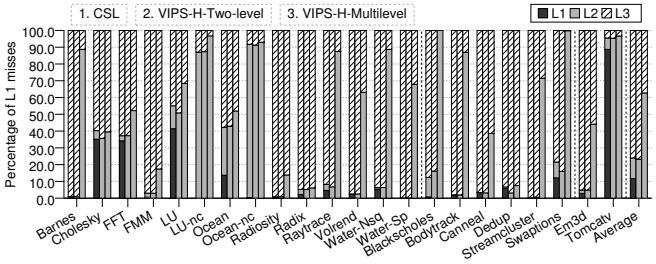
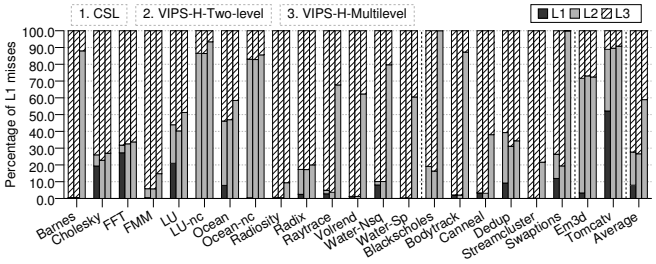


Fig. 5. Fraction of pages classified to each CSL



(a) 4×4 clustered system



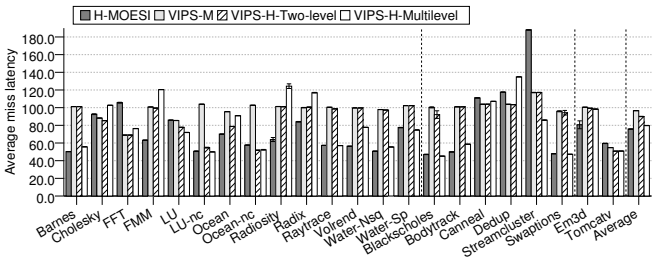
(b) 16×4 clustered system

Fig. 6. CSL of L1 misses and their resolution level (L2/L3)

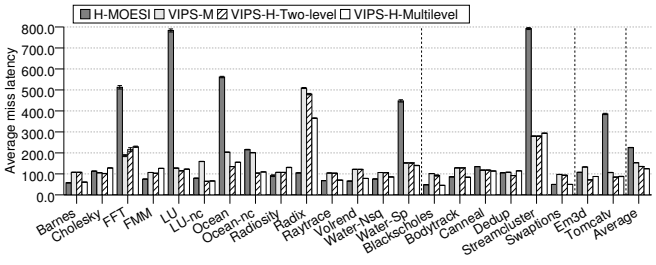
and multilevel policies (second and third bar, respectively). Between 70% and 80%, on average, of L1 misses are for blocks whose CSL is the L3. This fraction increases with respect to Figure 5 since private blocks suffer fewer misses than shared blocks. There is a high correlation between the CSL and the level of resolution of a miss for the two-level policy, since the L2 (i.e., the intermediate level in this case) does not hold blocks whose CSL is the L3. However, allowing caching of these blocks (multilevel policy) leads to better utilization of the L2, and as a consequence a reduction of the miss latency both in the 4×4 and the 16×4 configurations.

D. Performance comparison to H-MOESI and VIPS-M

Figure 7 shows the average L1 miss latency for every application evaluated, for H-MOESI, VIPS-M, VIPS-H-Two-level, and VIPS-H-Multilevel. H-MOESI does not scale well as it increases the average latency considerably: from 78 cycles in the 4×4 configuration to 220 cycles in the 16×4 configuration. This is mainly due to the *blocking* of cache controllers while performing invalidations and forwarding. Queuing latency in the cache controllers increases considerably with more cores (and levels) in the system. In contrast, the average latency in the VIPS protocols only increases from 80–100 cycles to 120–150 cycles. The exception is *Radix*, where the large number of L3 misses (see Figure 6) due to all-to-all communication [41] increases L3 contention, and therefore, miss latency.

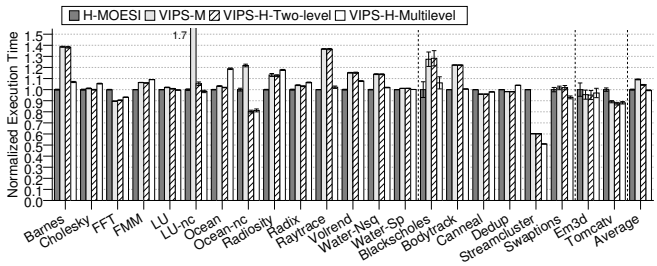


(a) 4×4 clustered system

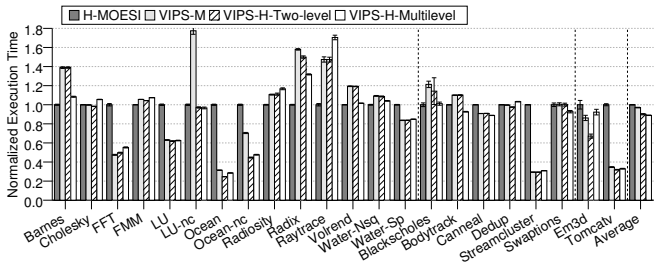


(b) 16×4 clustered system

Fig. 7. Average miss latency



(a) 4×4 clustered system



(b) 16×4 clustered system

Fig. 8. Execution time normalized to H-MOESI

Additionally, the two policies of VIPS-H reduce the latency with respect to VIPS-M for both the configuration with 16 cores and the configuration with 64 cores due to performing a more accurate classification, which allows the protocol to be more selective in the self-invalidation and to write-through to lower cache levels. Finally, VIPS-H-Multilevel reduces the latency with respect to VIPS-H-Two-level due to a better use of the L2 cache.

Figure 8 shows the application execution time of VIPS-M and VIPS-H normalized with respect to H-MOESI. On average, VIPS-M degrades performance compared to H-MOESI for the 4×4 configuration, but obtains similar performance for the 16×4 configuration. On the other hand, although VIPS-H does not obtain performance improvements with respect to H-MOESI for a 4×4 configuration, it achieves 10% *less* execution time for the 16×4 configuration. This indicates better scalability for the VIPS protocols in general and for VIPS-H in particular.

Some applications (*Radiosity*, *Radix*, and *Raytrace*) perform worse with VIPS-H than with H-MOESI for 16×4. As shown in Figure 7, VIPS-H considerably increases miss latency for *Radix* (a large percentage of misses go to the L3), with a direct impact on execution time. *Raytrace*, on the other hand, has a significant number of locks (94.5K [41]), which result in excessive self-invalidation, and thus, extra misses. These misses are evident in the *Response_data* bar in Figure 9 which grows considerably in *Raytrace*. The extra misses adversely impact execution time. The increase in the *Response_data* bar, is far worse in *Radiosity*. *Radiosity* has far more locks (231K [41]) than *Raytrace* and is one the most challenging programs to run in self-invalidation protocols. Curiously, its execution time is not as severely affected. The reason is that, while *Raytrace* spends very little time in synchronization (5%), *Radiosity* spends about 30% of its execution time! Its performance is dominated by synchronization overhead and the penalty of the extra self-invalidation misses is mitigated.

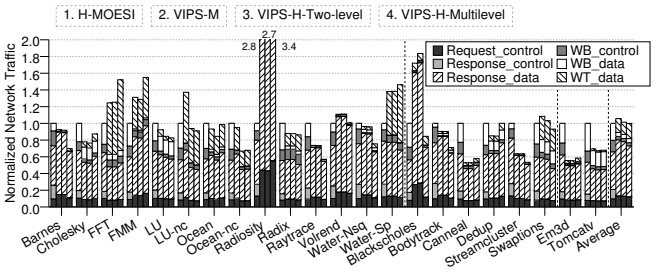
Regarding VIPS-H policies, some applications, like *Barnes*, *Volrend*, *Blackscholes*, and *Bodytrack* performs better with the multilevel than with the two-level policy for the 16×4 configuration, as expected. However, for some other applications, such as *Raytrace* and *Em3d* the two-level policy exhibits better performance. This happens in applications with intense locking, where self-invalidation is frequent and therefore caching at the L2 is not useful, and the effect of bypassing the L2 on every request to the L3 level can actually reduce miss latency.

The network traffic is shown in Figure 9, normalized to H-MOESI. On average, VIPS-M does not saves network traffic compared to H-MOESI since the self-invalidation of unnecessary cache blocks causes extra cache misses, and consequently extra traffic, as shown by the increase in the *Response_data* bar. VIPS-H *reduces traffic in 19 out of the 22 benchmarks*. This reduction in traffic is achieved despite the enormous traffic generated by *Radiosity*, which is the only outlier. Again, VIPS-H scales well with respect to H-MOESI, by reducing the traffic more for the 16×4 configuration (12%) than for the 4×4 one (7%). A key to this reduction is the low overhead entailed by the write-through traffic (*WT_data* bar in Figure 9). Most of the write-backs are converted to write-throughs, which send diffs across the network.

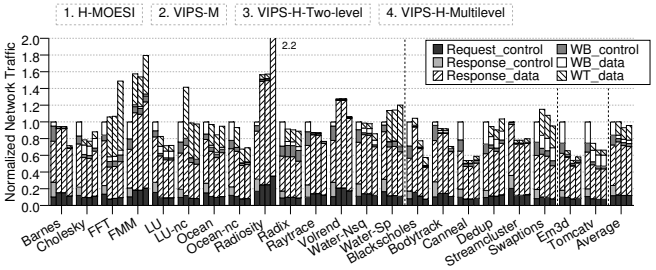
Traffic and execution time in VIPS-H may not be directly related. The reason is that write-throughs, which cause traffic, increase due to propagation of writes to higher cache levels, but that does not cause an increase in execution time as writes do not block the cores. Note that the increase in traffic in *fft* is due to WT-data.

E. Performance comparison to H-Token

To compare with H-token we employ the simple network model provided by GEMS instead of Garnet. The reason is that the miss latency variability introduced in Garnet renders the prediction to reissue requests inaccurate, thus injecting extra traffic and causing considerable congestion in the network and slowdown in the applications, especially for the 16×4 configuration. *Em3d* for the 4×4 system and *Em3d*, *Ocean*,

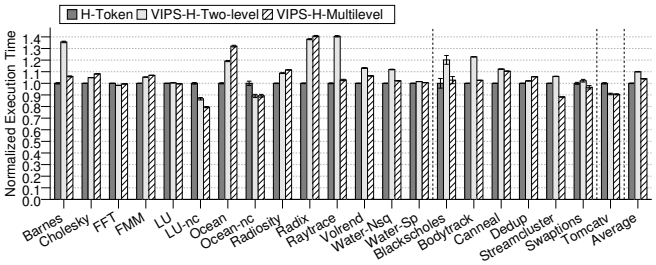


(a) 4×4 clustered system

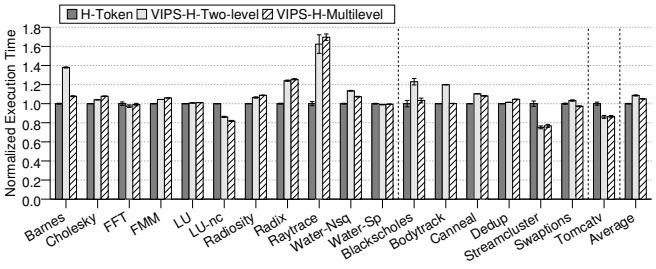


(b) 16×4 clustered system

Fig. 9. Network traffic normalized to H-MOESI



(a) 4×4 clustered system

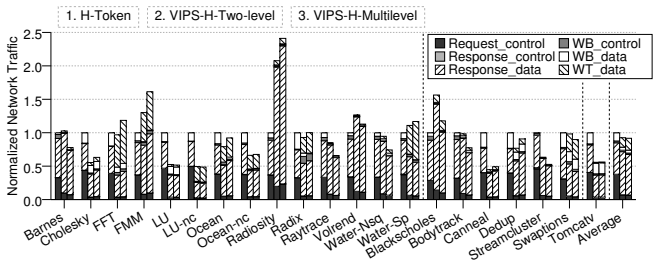


(b) 16×4 clustered system

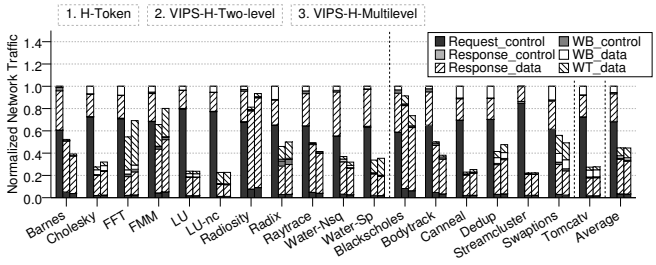
Fig. 10. Execution time normalized to H-Token

Ocean-nc, and *Volrend* for the 16×4 system where causing deadlocks when running the token protocol, and therefore, were excluded.

Figure 10 shows that H-Token edges VIPS-H slightly in execution time when assuming such an ideal network. In some benchmarks H-Token outperforms VIPS-H and in others vice versa. In 64 cores, execution time differences become smaller and the performance of VIPS-H converges to that of H-Token. VIPS-H, however, has the upper hand in network traffic in both the 16- and the 64-core case (8% less traffic for the 4×4 configuration and 55% for the 16×4 configuration). In fact, the differences in network traffic *increase significantly* in the larger system showing that VIPS-H scales better than H-Token.



(a) 4×4 clustered system



(b) 16×4 clustered system

Fig. 11. Network traffic normalized to H-Token

VII. Conclusions

In this work we set out to answer whether a simple and efficient approach to coherence is feasible for hierarchical clustered cache architectures. We show how this can be achieved by using simple mechanisms such as self-invalidation and write-through/self-downgrade, coupled with a hierarchical private/shared classification of data. The hierarchical private/shared classification encompasses the complexity of the hierarchy and allows simple implementations of far more common coherence operations. The end result is a coherence protocol that uses a fraction of the states (complexity) of a hierarchical directory protocol, at a comparable or better performance and reduced network traffic (12% overall and significantly reduced network traffic in 19 out of 22 benchmarks).

Acknowledgment

This work was supported by the "Fundación Seneca-Agencia de Ciencia y Tecnología de la Región de Murcia" under grant "Jóvenes Líderes en Investigación" 18956/JLI/13, and by by the Spanish MINECO, as well as European Commission FEDER funds, under grant TIN2012-38341-C04-03.

References

[1] J. A. W. Wilson, "Hierarchical cache/bus architecture for shared memory multiprocessors," in *14th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1987, pp. 244–252.

[2] S. V. Adve and M. D. Hill, "Weak ordering – a new definition," in *17th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1990, pp. 2–14.

[3] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 33–42.

[4] A. R. Alameldeen and D. A. Wood, "Variability in architectural simulations of multi-threaded workloads," in *9th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2003, pp. 7–18.

[5] M. Alisafaee, "Spatiotemporal coherence tracking," in *45th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2012, pp. 341–350.

[6] T. J. Ashby, P. Díaz, and M. Cintra, "Software-based cache coherence with hardware-assisted selective self-invalidations using bloom filters," *IEEE Transactions on Computers (TC)*, vol. 60, no. 4, pp. 472–483, Apr. 2011.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2008, pp. 72–81.

[8] M. Butler, L. Barnes, D. D. Sarma, and B. Gelinas, "Bulldozer: An approach to multithreaded compute performance," *IEEE Micro*, vol. 31, no. 2, pp. 6–15, Mar. 2011.

[9] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the memory hierarchy for disciplined parallelism," in *20th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2011, pp. 155–166.

[10] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *38th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2011, pp. 93–103.

[11] N. Eisley, L.-S. Peh, and L. Shang, "In-network cache coherence," in *39th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2006, pp. 321–332.

[12] N. D. Enright-Jerger, L.-S. Peh, and M. H. Lipasti, "Virtual tree coherence: Leveraging regions and in-network multicast tree for scalable cache coherence," in *41th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Nov. 2008, pp. 35–46.

[13] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *17th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2011, pp. 169–180.

[14] D. B. Gustavson, "The scalable coherent interface and related standards proyects," *IEEE Micro*, vol. 12, no. 1, pp. 10–22, Jan. 1992.

[15] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-optimal block placement and replication in distributed caches," in *36th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2009, pp. 184–195.

[16] H. Hossain, S. Dwarkadas, and M. C. Huang, "POPS: Coherence protocol optimization for both private and shared data," in *20th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2011, pp. 45–55.

[17] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-race-free memory models," in *19th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Feb. 2014, pp. 427–440.

[18] S. Kaxiras and J. R. Goodman, "The glow cache coherence protocol extensions for widely shared data," in *10th Int'l Conf. on Supercomputing (ICS)*, Jan. 1996, pp. 35–43.

[19] S. Kaxiras and G. Keramidas, "SARC coherence: Scaling directory cache coherence in performance and power," *IEEE Micro*, vol. 30, no. 5, pp. 54–65, Sep. 2011.

[20] S. Kaxiras and A. Ros, "A new perspective for efficient virtual-cache coherence," in *40th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2013, pp. 535–547.

[21] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Rigel: an architecture and scalable programming interface for a 1000-core accelerator," in *36th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2009, pp. 140–151.

[22] D. Kim, J. A. J. Kim, and J. Huh, "Subspace snooping: Filtering snoops with operating system support," in *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2010, pp. 111–122.

[23] E. Ladan-Mozes and C. E. Leiserson, "A consistency architecture for hierarchical shared caches," in *20th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, Jun. 2008, pp. 11–22.

[24] A. R. Lebeck and D. A. Wood, "Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors," in *22nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1995, pp. 48–59.

[25] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. L. Hennessy, M. A. Horowitz, and M. S. Lam, "The Stanford DASH multiprocessor," *IEEE Computer*, vol. 25, no. 3, pp. 63–79, Mar. 1992.

[26] Y.-C. Maa, D. K. Pradhan, and D. Thiebaut, "Two economical directory schemes for large-scale cache coherent multiprocessors," *ACM SIGARCH Computer Architecture News*, vol. 19, p. 10, Sep. 1991.

[27] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Communications of the ACM*, vol. 55, pp. 78–89, Jul. 2012.

[28] M. M. Martin, "Token coherence," Ph.D. dissertation, University of Wisconsin-Madison, Dec. 2003.

[29] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[30] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. Martin, and D. A. Wood, "Improving multiple-cmp systems using token coherence," in *11th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2005, pp. 328–339.

[31] M. R. Marty and M. D. Hill, "Virtual hierarchies to support server consolidation," in *34th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2007, pp. 46–56.

[32] M. R. Marty and M. D. Hill, "Virtual hierarchies," *IEEE Micro*, vol. 28, no. 1, pp. 99–109, Jan. 2008.

[33] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, Mar. 2010.

[34] H. Nilsson and P. Stenström, "The scalable tree protocol - A cache coherence approach for large-scale multiprocessors," in *4th Int'l Conference on Parallel and Distributed Computing*, Dec. 1992, pp. 498–506.

[35] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian, "SWEL: Hardware cache coherence protocols to map shared data onto shared caches," in *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2010, pp. 465–476.

[36] A. Ros, B. Cuesta, M. E. Gómez, A. Robles, and J. Duato, "Temporal-aware mechanism to detect private data in chip multiprocessors," in *42nd Int'l Conf. on Parallel Processing (ICPP)*, Oct. 2013, pp. 562–571.

[37] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 241–252.

[38] M. Shah, J. Barreh, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Saha, D. Sheahan, L. Spracklen, and A. Wynn, "UltraSPARC T2: A highly-threaded, power-efficient, SPARC SoC," in *IEEE Asian Solid-State Circuits Conference*, Nov. 2007, pp. 22–25.

[39] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, ser. Synthesis Lectures on Computer Architecture, M. D. Hill, Ed. Morgan & Claypool Publishers, 2011.

[40] H. Sung, R. Komuravelli, and S. V. Adve, "DeNovoND: Efficient hardware support for disciplined non-determinism," in *18th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2013, pp. 13–26.

[41] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *22nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1995, pp. 24–36.

[42] D. A. Wood, S. Chandra, B. Falsafi, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, S. S. Mukherjee, S. Palacharla, and S. K. Reinhardt, "Mechanisms for cooperative shared memory," in *20st Int'l Symp. on Computer Architecture (ISCA)*, May 1993, pp. 156–167.

[43] J. Zebchuk, B. Falsafi, and A. Moshovos, "Multi-grain coherence directories," in *46th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2013, pp. 359–370.

[44] M. Zhang, A. R. Lebeck, and D. J. Sorin, "Fractal coherence: Scalably verifiable cache coherence," in *43rd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2010, pp. 471–482.