Scope-Aware Classification: Taking the Hierarchical Private/Shared Data Classification to the Next Level

Mahdad Davari, Erik Hagersten, Stefanos Kaxiras

Department of Information Technology,
Uppsala University, Sweden
{mahdad.davari,erik.hagersten,stefanos.kaxiras}@it.uu.se

Abstract. Hierarchical techniques are commonplace in ameliorating the bottlenecks, such as cache coherence, in the design of scalable multi/many-cores. Furthermore, there have been proposals to simplify the coherence based on the data-race-free semantics of the software and private/shared data classification, where cores self-invalidate their shared data upon synchronizations. However, naive private/shared data classification in the hierarchies nullifies such optimizations by increasing the amount of data misclassified as shared and therefore being needlessly self-invalidated.

We introduce a private/shared data classification approach for hierarchical clusters, where a datum is concurrently classified as private and shared with respect to different classification scopes. Such scope-aware classification eliminates the needless self-invalidation of the valid data at synchronizations, resulting in a coherence scheme that reduces the average network traffic and execution time by 30% and 5%, respectively.

1 Introduction

Hierarchical and clustered design of the multi/many-core architectures are common practices to alleviate the storage and the communication bottlenecks of maintaining cache coherency [23, 16, 20]. Moreover, the data-race-free (DRF) semantics of today's programming models enables hybrid approaches that can be leveraged to simplify the hardware coherence itself [2, 32]: sharers need not be tracked and write-induced invalidations are eliminated as long as each core self-invalidates its own stale data upon the explicit synchronizations marked by *barrier* and *acquire/release* semantics, where the stale data are associated with the data classified as shared [26]. While self-invalidating the shared data in a private first-level (L1) cache only affects the core that is crossing a synchronization point, the penalty is exacerbated in the hierarchical architectures since self-invalidating the shared data in a shared large-capacity intermediate-level caches affects all the cores that share the cache.

This work aims to introduce a private/shared data classification scheme[1] that can be leveraged to enable efficient cache coherent schemes for the hierarchical

[1] By data we refer to the granularity at which coherence is typically maintained in multicores, i.e. a cache line.

topologies. Unlike a non-hierarchical scheme, where a datum is uniformly classified as private or shared in all the caches, a datum can be classified as private or shared in different caches in the hierarchical classification scheme. Our approach takes this a step further by introducing the notion of classification scopes, where in each cache a datum is concurrently classified as private and shared with respect to the classification scopes. Such scope-aware classification further enables scoped synchronization, which eliminates the needless self-invalidation of the shared data when the scope of the synchronization is different from the scope where the data are classified as shared.

2 Background

2.1 Hierarchical Coherence

Hierarchical coherence has been extensively studied by previous work [1, 18, 24, 17, 15]. The basic idea is to mitigate the coherence traffic by localizing memory accesses within the clusters, and to mitigate the coherence storage overhead by allowing coarse-grained inter-cluster sharer tracking [20], resulting in more scalability and lower formal-verification cost [21].

2.2 Optimizations based on Data Classification

Classification of data into private and shared has been leveraged to enable optimizations in cache and coherence design, such as efficient data placement policies for private and shared data in memory hierarchies [12], or deactivation of coherence for the private data in order to reduce the directory size [7, 4, 34].

2.3 Coherence for Data-Race-Free Semantics

Previous work advocates the benefits of the DRF programming models for simplifying the cache coherence [32, 2]. DRF semantics divides the execution of a program into epochs, explicitly marked by synchronizations, where in any given epoch i) at most one thread shall modify a datum, and ii) no threads—except the modifying thread—shall read a datum if the datum is modified in that epoch. In other words, DRF semantics shifts the burden of maintaining the *single-writer-multiple-readers* invariant from the hardware to the software [29]. As a result, the directory does not need to track the readers in order to send invalidations when a thread is about to modify the shared datum, as long as each core self-invalidates its stale data at the beginning of each epoch. DeNovo-family of coherence protocols [6, 31, 30] and VIPS-family of coherence protocols [27, 26, 8, 9] are examples of DRF coherence schemes.

2.4 Scoped Synchronization

The notion of *synchronization scopes* has been studied in the context of the general purpose GPUs (GPGPUs) [14, 25, 10]. The idea is to avoid the needless

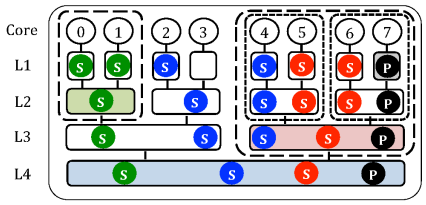


Fig. 1: Non-hierarchical classification applied to a hierarchy (P:Private, S:Shared).

global synchronizations by confining the synchronizations to the local scopes in the memory hierarchy. However, the memory model is complicated due to the aditional scope-related semantics. While solutions have been proposed to achieve similar benefits for the GPGPUs without complicating the memory model [28], our scope-aware classification and synchronization extends the concept to the more general hierarchical and clustered CPU architectures.

3 Hierarchical Private/Shared Data Classification

3.1 System Topology and Terminology

While our approach is topology-agnostic, we assume symmetric, fully-populated, hierarchical clusters as shown in Fig.1 in order to explain the classification.

L1 caches in Fig.1 are referred to as sources, where initial requests originate upon L1 cache misses. Caches between L1s and the LLC are referred to as intermediate caches. A cluster, marked by the dashed lines, represents an entity comprised of sources and a hierarchy of shared intermediate caches. The highest intermediate cache in a cluster, which is shared by all the sources in that cluster, is referred to as the sink. A cluster is represented by its sink. As an example, the leftmost L2 cache in Fig.1 is the sink for the cluster encompassing sources 0 and 1, and the rightmost L3 cache is the sink that represents the cluster encompassing sources 4 to 7. Similarly, the LLC is considered as the global sink that represents the global cluster (super-cluster) encompassing all the clusters (sub-clusters.) Moreover, each cluster defines a scope that covers the sources, the sink, and all the shared intermediate caches.

3.2 Hierarchical Classification

A non-hierarchical data classification scheme applied to hierarchical cache clusters is shown in Fig.1. While uniformity of classification across the hierarchy enables less complex classification mechanisms, opportunities for optimizations based on locality of data are lost [12, 7]. For instance, green data in Fig.1 are shared between cores 0 and 1, and therefore are considered as shared inside the scope represented by the green sink. However, same data are considered as private anywhere outside that scope. The ultimate goal of the hierarchical

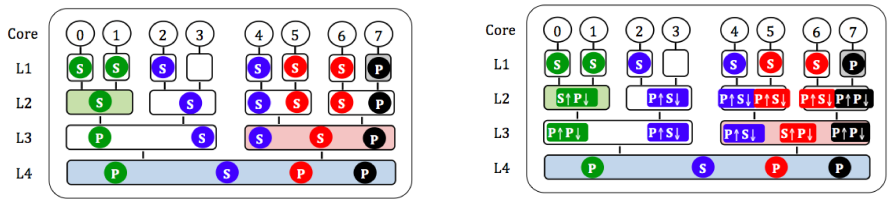
private/shared classification, shown in Fig.2a, is to preserve the private classification in the hierarchies. The first step is to incorporate the notion of scopes into the hierarchical data classification.

Classification is performed at the sinks. The first data request does not find the cache line in any sink until it reaches the LLC, where the cache line is classified as private owned by the requesting source. The LLC responds to the sink of the sub-cluster that issued the request. Data and classification are recursively routed backwards until they are received by the source of the request. Classification is abstracted as a tuple that consists of a single private/shared bit and an *owner* field. The owner-field need not be wide enough to track the sources. We require coarse-grained tracking at the intermediate caches. Therefore, the LLC shown in Fig.2a only needs to know which of the left or right sub-clusters owns the data, reducing the storage to a third compared to full-map tracking in the given example. While the classification metadata can be integrated into the sinks [8], we opt for a minimal directory cache [9].

Re-adjustment of the classification is required when a sink receives a request for a cache line that is already classified as private. A *Private-to-Shared* (hereafter P2S) transition [9] is activated, which probes the private owner to find out whether data are still held by the owner. The P2S probe is recursively routed in the hierarchy until the probe is received by the source. In case the source has silently evicted the data—section 4.1,—data are reclassified as private with a new source. Otherwise, the source changes its internal classification to shared and responds with up-to-date data—in case it has modified the data. On its way towards the sink, the probe response re-adjusts the classification in all the intermediate caches. As we show in section 4.1 and explain the write-policy for the shared data, classification re-adjustment in the intermediate caches is required to guarantee that sinks always contain up-to-date data for the future requests. After probe response is received, classification re-adjustment is complete and current and future requestors receive the data classified as shared. In sections 4.2 and 5 we evaluate an alternative inclusion policy for the shared data that does not require shared data to be stored in the intermediate caches.

Fig.2a shows the aforementioned plain hierarchical classification applied to four cache lines. In contrast to the non-hierarchical classification (Fig.1,) the green data are classified as private inside scopes represented by sinks at third and fourth levels. The red data are also classified as private inside the scope represented by the sink at fourth level. However, the blue data in intermediate caches at second and third levels are classified as shared, although they are private inside the corresponding scopes. The same holds true for the red data in scopes represented by sinks at second level. In other words, external scopes override the locality of classification for the local scopes. Can this lead to inefficiency and performance loss?

To answer this question, a scenario can be assumed where core 4 in Fig.2a is synchronizing with core 2. In order to make sure core 4 does not access stale data after synchronization, core 4 self-invalidates data that might be modified by core 2, which corresponds to shared data in all the sinks up to the LLC. Apart from



(a) Plain hierarchical classification. (b) Scope-aware classification.

Fig. 2: Hierarchical data classification.

the shared blue data, which are accessed by core 4, the shared red data at the L2 sink are also invalidated, although the red data are locally private to core 5 and might be valid and in active use by core 5. The root of this problem lies in (i) inability of the classification to preserve the locality of classification inside each scope, and (ii) inability of the synchronizations to detect and invalidate only the shared data that belong to the source issuing the synchronization. To address these problems, we introduce the scope-aware classification that preserves the locality of the classification inside the scopes.

3.3 Scope-Aware Classification

Scope-aware classification extends the plain hierarchical classification (Sec.3.2) so that the externally shared data can be locally classified as private.

We focus on the cases where the plain hierarchical classification was unable to preserve the locality of classification within the scopes. Fig.2 shows both the plain and the scope-aware classification for the same data. The classification for the shared data colored red at the L2 sinks now captures both local and external scopes. The first element (preceded by an upward-pointing arrow) corresponds to the local-scope inside the cluster encompassing the cores 4 and 5. Since the red data are only accessed by core 5 in the mentioned scope, the data are locally classified as private. However, the same data are shared when seen from the external scope represented by the sink at the L3, since the data exist in two sub-clusters seen from the sink at the L3. As a result, the same data are classified as shared in the external scope (preceded by a downward-pointing arrow). On the other hand, the red data at L3 are locally shared inside the scope represented by the sink at the L3, while the same data are private at the external scope represented by the LLC, since the data exist only in one sub-cluster seen from the sink at LLC. Similarly, the green data at the L2 are locally classified as shared and externally classified as private. The green data at the L3 however are private in both local and external scopes when observed from the L3. Furthermore, the blue data are recursively classified as locally-private/externally-shared in the hierarchy before reaching the LLC. As shown in the Fig.2b, scope-aware classification correctly detects the local and external scopes and performs the

appropriate private/shared classification accordingly. Finally, the data in black are classified as private at all the scopes.

Classification storage: scope-aware classification has the similar storage requirements as the plain hierarchical classification (Sec. 3.2. In addition to that, an extra bit per cache line is required to store the external-scope classification.

P2S transition: the same mechanisms explained in Sec. 3.2 are used. However, external-scope classification is used to decide whether the P2S transition should be invoked. Furthermore, the recursive routing of the P2S probe to the source might be terminated before the probe reaches the source, if the P2S probe arrives at a sink where data are locally classified as shared—green data in L2 in Fig.2b. Moreover, on its way towards the sink, the probe response readjusts the external-scope classification of intermediate caches. As an example, although the red data in L2 shown in Fig.2b are locally classified as private to core 5, P2S operation is not required when the red data are accessed by core 4. Since the red data are externally classified as shared in L2, core 5 has already made P2S transition and therefore self-invalidates its copy of red data in its L1 cache upon synchronization. As a result, local-scope classification for the red data at L2 is silently changed to shared when the red data at L2 are accessed by core 4. However, access to black data by a core other than core 7 invokes the P2S transition across all the intermediate caches.

3.4 Scoped Synchronization

Having two distinct local and external classification leaves the reader with the question how the data should be self-invalidated with respect to these scopes? At each intermediate cache, the data that are externally classified as private are exempt from self-invalidation, regardless of being classified locally as private or shared, since such data are not externally modified due to being classified as private at external scope. The data that are classified as shared at the external scope are likely to be externally modified in the current phase, and are therefore required to be self-invalidated before crossing a synchronization point. However, consider the case in Fig.2b where core 5 and core 6 are crossing a synchronization point for the shared red data. A self-invalidation request from core 5 is sent to the L2, where both externally-shared blue and red data reside. The blue data are locally classified as private to core 4 and the red data are internally classified as private to core 5. Self-invalidating the externally-shared blue data is redundant in this case since core 5 is synchronizing for the red data only, and core 4 may lose its valid blue data which are frequently accessed. Self-invalidating all the externally-shared data is therefore not required, and can be mitigated through scoped synchronization, which requires that the externally-shared locally-private data be self-invalidated only if the synchronizations come from the owners of those data in the local scope. The data that are classified as shared both at the local and external scopes are nevertheless required to be self-invalidated, since sharers are not tracked and therefore relations cannot be established between the synchronizations and the intended data sets. On the other hand, in the aforementioned example core 5 may also need to access the blue data after the

Classification	Response to Synchronization
locally-private, externally-private	Insensitive
locally-private, externally-shared	self-invalidates upon synchronizations by owner
locally-shared, externally-private	Insensitive (sinks always contain valid data)
locally-shared, externally-shared	Always self-invalidates

Table 1: Self-invalidation based on scoped Synchronization.

synchronization. Core 5 is likely to receive the stale data from the L2 cache, since the blue data is not self-invalidated due to the scoped synchronization. In order to address this issue, scoped synchronization requires that accesses to the valid data classified as <locally-private, externally-shared>by cores other than the owner in the local scope should read the value through the higher shared caches (e.g. the sink or the LLC) where the valid value exists. Once the values are read through, the data are re-classified as <locally-shared, externally-shared>, which makes the data susceptible to all the future synchronizations.

As the previous example showed, the scoped synchronization aims to mitigate the penalty of the self-invalidations in the intermediate shared caches in the hierarchies. Instead of bulk self-invalidation of all the shared data, only the data that belong to the core crossing the synchronization are self-invalidated. Nevertheless, the data are read-through on demand, which translates into more selective self-invalidations. Furthermore, the scoped synchronization does not rely on the software to obtain information about the scopes, thus not complicating the memory model. The scope of the synchronization is dynamically detected in the hardware based on the identity of the core issuing the synchronization and the internal and external scopes of the classification.

4 DIR_1-H: Coherence for Clustered Hierarchies

4.1 Implementation and Mechanisms

We propose the *DIR_1-H* hierarchical coherence scheme based on the scope-aware classification and the previous work *Dir_1-SISD* [9]. Dir_1-H employs the following mechanisms to ensure that valid data are found in the sinks:

Self-Invalidation: L1s are required to self-invalidate their shared data upon synchronization. Intermediate caches up to LLC self-invalidate their data based on the scope of classification and synchronization according to Table 1.

Self-Downgrade: L1s self-downgrade their shared data by writing through to their next shared cache in the hierarchy. Intermediate caches write through to higher levels—up to the sink—only if data are externally classified as shared, regardless of the local classification.

4.2 Optional Inclusion Policies for Shared Data

The inclusion policy for the shared data in the hierarchy determines the complexity of self-invalidations and self-downgrades. Mechanisms discussed so far concern

Parameter	Value
Processor frequency	3.0GHz, In-Order
Block(Page) size/MSHR size	64(4K) bytes/16entry(1000cycles delay-timeout)
Split L1 I/D caches size/hit latency	32KB, 4-way/1(tag) + 1(data) cycles
L2 cache size/hit latency	4MB, 16-way/6(tag) + 6(data) cycles
L3 cache size/hit latency	16MB, 32-way/10(tag) + 20(data) cycles
L2 directory-cache	1024 entries per L1 (size factor 2)
Flit/Message size	16bytes/data:72bytes(5flits);control:8bytes(1flit)
Memory access/switch2switch time	160cycles/6(13) on-(off-)chip cycles

Table 2: Base system parameters

an inclusive hierarchy. Similar to the private data, the shared data are replicated in all the levels. Although the miss latency is reduced, self-invalidations and self-downgrades become more complex as they need to traverse the hierarchy towards the LLC. We design and evaluate a version of Dir_1-H protocol that relaxes inclusion for shared data in the intermediate levels (see Sec.5.4.)

5 Evaluation

5.1 Setup

We compare the Dir_1-H coherence protocol (Sec.4.1) with i) hierarchical directory-based MOESI protocol distributed with GEMS simulator [22], and ii) hierarchical VIPS-H protocol [26]. We choose MOESI protocol as our baseline. VIPS-H relies on the operating system for page classification and uses similar self-invalidation/downgrade mechanisms. We also evaluate different inclusion policies for the shared data (Sec. 4.2): Dir_1-H-Incl and Dir_1-H-Excl employ inclusive and exclusive policy for the shared data, respectively.

We use the Simics full-system simulator [19],the Ruby memory-system simulator [22], and the GARNET network simulator [3]. We use the setup used by our baseline system: a hierarchical CPU system with three-levels including 16 cores, organized in 4 clusters of 4 cores each (4x4). While L1 caches are private to corresponding cores, L2 cache is partially shared by L1s within each cluster, and L3 cache is shared globally. Table 2 lists more details of our chosen setup.

We employ a wide variety of parallel applications (*SPLASH-2* [33] and *PARSEC* [5].) We simulate the entire applications, but collect statistics only during the parallel regions. Applications from the *SPLASH-2* use the *recommended* inputs, whereas the applications from *PARSEC* use *simmedium* inputs, except for the *Swaptions* that uses the *simsmall*.

5.2 Complexity and Cost

Dir_1-H protocol requires 12 states for the three levels of caches, whereas MOESI and VIPS-H require 88 and 18 states, respectively. MOESI is implemented as a

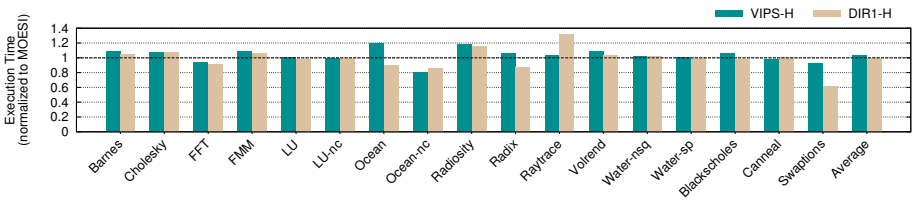


Fig. 3: Dir_1-H execution time normalized to VIPS-H.

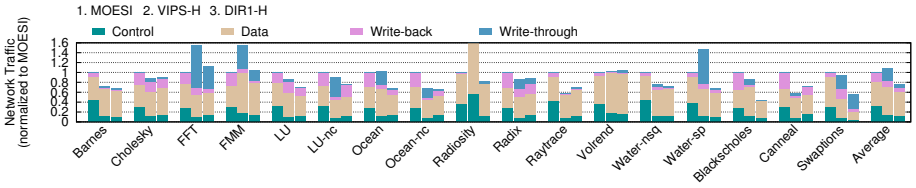


Fig. 4: Dir_1-H detailed network traffic normalized to VIPS-H. Radiosity stops at 3.

full-map in-cache directory. VIPS-H requires a single bit per L1, L2, and L3 cache line to hold the private/shared classification. Dir_1-H relaxes the requirement for the private/shared bit per L2 and L3 cache lines by holding the bits in the directory, resulting in less overheads due to using a sparse directory [11].

5.3 Performance

As shown in Figure 3, the three protocols perform similarly on average. VIPS-H classifies data at page granularity, which results in non-aligned private data to be misclassified as shared. If those data are frequently accessed, the self-invalidation penalty would be high in the presence of frequent synchronizations. The increase in execution time for Raytrace and Ocean-nc with Dir_1-H protocol is attributed to the directory evictions due to the limited size of the used sparse directory. VIPS-H is immune to such penalties, since it relies on operating system to store the classification of pages in the page-table entries.

As shown in Figure 4, Dir_1-H reduces the average network traffic by 23% compared to MOESI via eliminating the coherence invalidations. Dir_1-H also reduces the traffic by 30% compared to VIPS-H, by preserving the locality of the classification, which significantly reduces needless self-invalidations and self-downgrades. The increase in the traffic for Canneal is not in the critical path of execution (as in Raytrace,) but due to the increase in the write-back traffic. Dirty blocks constitute majority of the replacements for Canneal, which requires the whole private cache-line be written back, instead of only the modified data. Furthermore, Radiosity has the highest number of locks—231K—in the *SPLASH-2* and spends 30% of the execution in synchronizations, which translates into frequent synchronizations and frequent invalidation of the shared data.

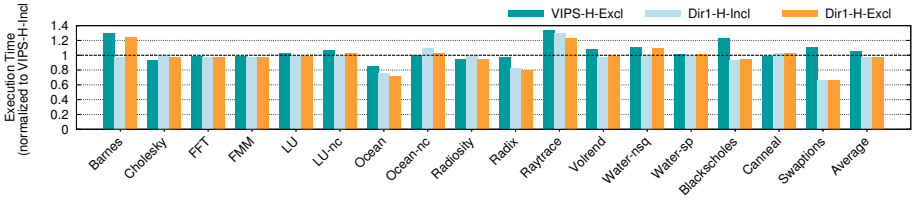


Fig. 5: Impact of the inclusion policy for the shared data on execution time.

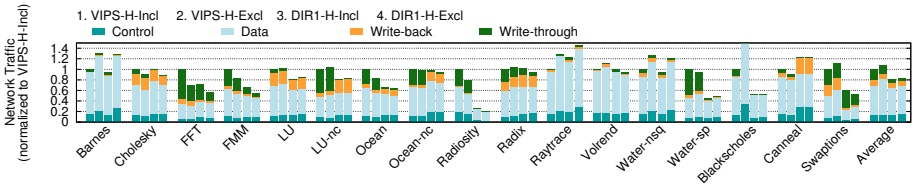


Fig. 6: Impact of the inclusion policy on the network traffic. Blackscholes stops at 2.

5.4 Impact of the inclusion policy for the shared data

The shared data are cached in the intermediate caches in VIPS-H-Incl and DIR_1-H-Incl, whereas VIPS-H-Excl and DIR_1-H-Excl only cache the shared data at L1 and sinks. While the former approach opts for minimizing miss latency, the latter aims at simplification of the self-invalidation/downgrade. Figures 5 and 6 suggest that requiring inclusion for shared data favors reduction in the network traffic and execution time for both VIPS-H and DIR_1-H protocols.

6 Conclusion

In this work we introduced a scope-aware private/shared data classification scheme for hierarchical CPU clusters. Our approach dynamically detects the scopes at which data are being classified. As a result, the same data can concurrently exist with different classification in different locations in the hierarchy, depending on the scope of the classification. Having both local and external classification for a datum at each scope enables scoped synchronization, which allows fine-grained and selective self-invalidation of externally shared data instead of needless bulk self-invalidations. Our results reveal that hierarchical coherence based on scope-aware classification and synchronization reduces the average network traffic and execution time by about 30% and 5% respectively, compared to a hierarchical coherence protocol based on a non-hierarchical data classification.

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. Adve and M. Hill: Weak ordering. 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. M. Alisafaee: Spatiotemporal coherence tracking. In *45th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2012, pp. 341–350.
5. 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.
6. 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.
7. 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.
8. M. Davari, A. Ros, E. Hagersten, and S. Kaxiras: The effects of granularity and adaptivity on private/shared classification for coherence. *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 3, Oct. 2015. http://dx.doi.org/10.1145/2790301
9. M. Davari, A. Ros, E. Hagersten, and S. Kaxiras: An efficient, self-contained, on-chip directory: Dir1-sisd. In *24th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2015, pp. 317–330.
10. B. R. Gaster, D. Hower, and L. Howes: Hrf-relaxed: Adapting hrf to the complexities of industrial heterogeneous memory models. *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 1, pp. 7:1–7:26, 2015. http://doi.acm.org/10.1145/2701618
11. A. Gupta, W.-D. Weber, and T. C. Mowry: Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Int'l Conf. on Parallel Processing (ICPP)*, Aug. 1990, pp. 312–321.
12. 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.
13. M. D. Hill and A. J. Smith: Evaluating associativity in CPU caches. *IEEE Transactions on Computers (TC)*, vol. 38, no. 12, Dec. 1989, pp. 1612–1630.
14. 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.
15. 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.
16. 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.
17. 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, Mar. 1992, pp. 63–79.

18. Y. Maa, D. Pradhan, and D. Thiebaut: Two economical directory for large-scale multiprocessors. *ACM SIGARCH Computer Architecture News*, vol. 19, Sep. 1991, p. 10.
19. P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner: Simics: A full system simulation platform. *IEEE Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.
20. 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.
21. M. M. Martin, M. Hill, and D. Wood: Token coherence: Decoupling performance and correctness. In *30th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2003, pp. 182–193.
22. 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.
23. J. Nickolls and W. J. Dally: The GPU computing era. *IEEE Micro*, vol. 30, no. 2, pp. 56–69, Mar. 2010.
24. 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.
25. M. S. Orr, S. Che, A. Yilmazer, B. M. Beckmann, M. D. Hill, and D. A. Wood: Synchronization using remote-scope promotion. In *20th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2015, pp. 73–86.
26. A. Ros, M. Davari, and S. Kaxiras: Hierarchical private/shared classification: key to simple coherence for clustered hierarchies. In *21st Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2015, pp. 186–197.
27. 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.
28. M. D. Sinclair, J. Alsop, and S. V. Adve: Efficient gpu synchronization without scopes: Saying no to complex consistency models. In *48th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2015, pp. 647–659.
29. D. J. Sorin, M. D. Hill, and D. A. Wood, *Primer on Memory Consistency and Cache Coherence*, M. D. Hill, Ed. Morgan & Claypool Publishers, 2011.
30. H. Sung and S. V. Adve: Denovosync: Efficient support for arbitrary synchronization without writer-initiated invalidations. In *20th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2015, pp. 545–559.
31. H. Sung, R. Komuravelli, and S. Adve: DeNovoND: 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.
32. L. Valiant: Bridging model for parallel computation. *Communications of the ACM*, vol. 33, pp. 103–111, Aug. 1990.
33. 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.
34. 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.