The Effects of Granularity and Adaptivity on Private/Shared Classification for Coherence

MAHDAD DAVARI, Uppsala University
ALBERTO ROS, Universidad de Murcia
ERIK HAGERSTEN, Uppsala University
STEFANOS KAXIRAS, Uppsala University

Classification of data into private and shared has proven to be a catalyst for techniques to reduce coherence cost, since private data can be taken out of coherence, and resources can be concentrated on providing coherence for shared data. In this paper we ask the question: how granularity—page-level vs. cache-line level—and adaptivity—going from shared to private—affect the outcome of classification and what is its final impact on coherence? To answer this, we create a classification technique, called *Generational Classification*, and a coherence protocol called *Generational Coherence,* which treats data as private or shared based on cache-line generations. We compare two coherence protocols based on self-invalidation/self-downgrade with respect to data classification. Our findings are enlightening: (i) Some programs benefit from finer granularity, some benefit further from adaptivity, but some do not benefit from either. (ii) Reducing the amount of shared data has no perceptible impact on coherence misses caused by self-invalidation of shared data, hence no impact on performance. (iii) In contrast, classifying more data as private has implications for protocols that employ write-through as a means of self-downgrade, resulting in network traffic reduction—up to 30%—by reducing the write-through traffic.

Categories and Subject Descriptors: **C.1.2. [Processor Architectures]**: Multiple Data Stream Architectures (Multiprocessors)

General Terms: Design, Performance

Additional Key Words and Phrases: Multicore, memory hierarchy, cache coherence

ACM Reference Format:

Mahdad Davari, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras, 2015. The effects of granularity and adaptivity on private/shared classification for coherence.

1. INTRODUCTION

The goal of simplifying coherence in multicore architectures is twofold: (i) to reduce the costs associated with coherence (area, energy, performance) and (ii) to increase scalability. It further enables diverse accelerator architectures and general purpose CPUs to seamlessly interconnect under coherent shared virtual memory. To this end, several recent proposals aim to reduce directory or snooping cost [Alisafaee 2012; Kim et al. 2010; Ferdman et al. 2011; Cuesta et al. 2011], while others simplify coherence by removing the directory bottleneck in its entirety [Choi et al. 2011; Ros and Kaxiras 2012; Pugsley et al. 2010; Hossain et al. 2011]. A common useful tool used by all such approaches is the classification of data into private and shared [Choi et al. 2011; Ros and Kaxiras 2012], resulting in directory size reduction [Pugsley et al.

Author's addresses: M. Davari, E. Hagersten and S. Kaxiras, Information Technology Department, Uppsala University; A. Ros, Computer Engineering Department, University of Murcia.

DOI:http://dx.doi.org/xx.xxxx/xxxxxxx.xxxxxxx

2010; Hossain et al. 2011; Alisafaee 2012], or optimizing the coherence protocol itself by replacing explicit coherence invalidations with silent self-invalidation of shared data upon synchronization [Ros and Kaxiras 2012].

In increasing order of hardware cost, data classification can be performed in three ways: (i) by the compiler [Li et al. 2010; Li et al. 2012], (ii) by the operating system [Hardavellas et al. 2009; Kim et al. 2010; Cuesta et al. 2011], or (iii) by hardware mechanisms [Pugsley et al. 2010; Hossain et al. 2011]. Although incurring minimum hardware cost, compiler-driven classification is not transparent to software. It is well-suited for hardware/software co-design, but requires re-coding and re-compilation for legacy software and significant effort to determine at compile time if a variable is going to be shared or not. The OS approach does not impose extra requirements for dedicated hardware, since data classification at page granularity is stored along with the page table entries (PTEs) [Hardavellas et al. 2009]. This makes it a good choice for complexity-effective optimizations. However, it suffers from granularity and adaptation problems, leading to misclassifications that increase with time, thus degrading the quality of classification. OS-based approaches are not limited to data classification for coherence. For instance, Fensch [Fensch et al. 2008] proposes a coherence scheme for tiled chip multiprocessor (CMPs) in which coherence task is divided between the OS and hardware. In their scheme, incoherence is avoided by not allowing data to be replicated across the tiles. This is achieved by having the OS to map pages to separate private level-one caches (hereafter L1) and having the hardware to allow a limited and controlled data migration via remote cache accesses. Although not affecting the application software, such approaches heavily depend on the OS for data mapping and movement in the system. Finally, hardware mechanisms work totally transparent to software, at page or cache line granularity, but can have prohibitive storage requirements [Pugsley et al. 2010; Hossain et al. 2011] or dual-granularity complexities [Alisafaee 2012; Zebchuk et al. 2013].

In this paper we examine the impact of classification when it is applied to coherence. We examine the case where private/shared classification is employed to simplify coherence by excluding private data from any coherence mechanism and employing simple mechanisms such as write-through and self-invalidation for the shared data. Such an approach has been proven to be effective using page-level, non-adaptive classification [Hardavellas et al. 2009; Ros and Kaxiras 2012]. Our aim is to assess the effect of finer granularity and adaptivity in the quality of the classification and ultimately in the performance of coherence. To this end, our first contribution is a robust approach for adaptive data classification at cache-line granularity that aims to address the weaknesses of the popular page-based OS approach, and be more accurate and efficient than prior hardware proposals [Alisafaee 2012; Zebchuk et al. 2013]. We call our classification *Generational* since we use a precise definition, based on the *generational behavior* of cache lines [Wood et al. 1991] (section 3). Prior works [Alisafaee 2012; Zebchuk et al. 2013] employ complex dual-grain adaptive mechanisms to detect private and shared regions, with the goal of reducing the directory size in mind. In contrast, our goal is to provide a robust classification to support coherence protocols that critically depend on the separation of data into private and shared. While prior methods require a full-blown directory to implement data classification, we perform the classification entirely in last-level cache (hereafter LLC), requiring a minimum amount of storage to hold the classification information. Such storage is easily integrated into LLC tags as attribute bits. We further develop a coherence protocol, called *Generational Coherence*, which combines generational data classification with self-invalidation and self-downgrade [Ros and Kaxiras 2012; Choi et al. 2011] (Section 4). In the rest of the document we use *GC* to refer to both *Generational Classification* and *Generational Coherence*, as they are tightly coupled.

Our second contribution is to compare GC with a page-based, non-adaptive protocol (Section 6). We examine three metrics: the quality of classification, the effect on self-invalidation, and the effect on write-through traffic. Our findings reveal that: (i) GC significantly increases the amount of private data, but there is also a class of programs where page-based, non-adaptive, classification already works well. (ii) Even when GC manages to classify significantly more data as private, the end effect on misses caused by self-invalidation is negligible. (iii) The effect on write-through traffic, however, is significant, leading to benefits in overall network traffic and consequently lower energy consumption. Further, we found that attempts to modulate the generational behavior using cache decay as a simple dead-block predictor are ineffective, given the time scale on which the classification operates.

2. BACKGROUND AND RELATED WORK

2.1 VIPS-M

A number of proposals recently advocate simple coherence for data-race-free semantics [Choi et al. 2011; Ros and Kaxiras 2012; Kaxiras and Ros 2012; Kaxiras and Ros 2013]. Due to its simplicity and being transparent to software, we have modeled our proposed generational coherence after VIPS-M coherence protocol [Ros and Kaxiras 2012; Kaxiras and Ros 2012; Kaxiras and Ros 2013]. VIPS-M is a directory-less protocol that relies on the properties of relaxed memory consistency and data-race-free semantics to allow incoherence in between synchronization points, but sequential consistency for data-race-free programs [Adve and Hill 1990; Sorin et al. 2011]. By relying on the data-race freedom property of programs, coherence among private L1s is maintained by three mechanisms: (i) a data classification mechanism that classifies data as private or shared, (ii) self-invalidation, and (iii) self-downgrade.

2.1.1. Private/Shared Data Classification. VIPS-M employs a technique that classifies data at a page granularity using the OS and the TLBs [Cuesta et al. 2011; Hardavellas et al. 2009; Kim et al. 2010]. A page accessed by a single core starts as private in the page table. Upon an access to the same page by a second core, the page becomes shared. A page becomes shared even when cores access entirely different cache lines in the page. When a page transitions from private to shared, the core that had previously tagged the page as private must be notified, so it can change its local classification of the page to shared. This is necessary since the write policy that is employed for the cache lines of a page is dictated by the local classification. A page transition from private to shared can happen at most once per page and a shared page never reverts back to private. VIPS-M also uses the same OS classification technique to distinguish between *Read-Only* pages and pages that are *Read/Write*. Cache lines belonging to Read-Only pages are not self-invalidated at synchronizations.

2.1.2. Self-Invalidation. Each L1 cache self-invalidates its shared data at synchronization points—lock acquisition or barrier crossing. As a result, all the explicit coherence invalidation traffic towards the L1s is eliminated.

2.1.3. Self-Downgrade. Self-downgrade is used by each L1 cache to keep the shared LLC up-to-date with the latest copy of the modified shared data in L1 caches. Prior work [Somogyi 2014; Somogyi et al. 2014] leverage self-downgrade technique to mitigate the coherence latency. Such techniques perform self-downgrade by implementing accurate but complex dead-block predictors. VIPS-M, on the other hand, introduces an efficient form of self-downgrade, whereupon written data are deliberately written through to LLC before any further access to them by other cores. This obviously eliminates forwarding and cache-to-cache transfers. The key observation that makes a write-through policy practical is that most write misses in a write-through protocol actually come from private blocks. Based on this observation, VIPS-M uses a dynamic write policy in the L1s: write-through for data that are *shared*, and write-back for *private* data that do not need coherence. As a further optimization, a write-through buffer is used to coalesce writes to the same cache lines. This buffer is emptied at the same synchronization points that cause self-invalidation. Write-throughs transfer only what is modified in a cache line—i.e. a *diff*. This allows multiple simultaneous writers to co-exist, with the guarantee of data-race-free property at word—or byte—level [Ros and Kaxiras 2012]. Self-invalidation and self-downgrade eliminate the need for a directory, since neither the writers—because of self-downgrade—nor the readers—because of self-invalidation—need to be tracked anymore.

2.2 Hardware private/shared classification approaches

Many systems rely on a hardware private/shared classification for a variety of reasons. Chief among these reasons is the directory size reduction [Alisafaee 2012; Zebchuk et al. 2013; Fang et al. 2013]. The approach is to implement a *multi-grain* directory that tracks coherence information on more than one block size. Alisafaee [Alisafaee 2012] and Zebchuk [Zebchuk et al. 2013] propose similar approaches that differ on implementations. The idea is to store private regions in the directory and out of those regions extract cache lines that are shared. Similarly, Fang [Fang et al. 2013] describes an approach where region entries can be private but block entries extracted from these regions can either be private—with a different private owner than the region— or shared—accessed by multiple cores.

Regarding adaptivity of classification, Pugsley [Pugsley et al. 2010] introduces an adaptive version of their SWEL protocol, called Reconstituted SWEL (RSWEL) that includes a 2-bit saturating decay counter to re-classify shared pages to private. However, the adaptation is quite complex, requires elaborate tuning of the time period that ticks the decay counters for good results, and lastly it is only initiated periodically in bulk. Furthermore, Zhao [Zhao et al. 2013] addresses adaptivity in cache coherence by proposing the Protozoa coherence protocol. However, Protozoa addresses adaptivity when applied to the granularity of coherence and data movement. Instead of having fixed cache-line granularity for both data storage/movement and coherence actions, Protozoa adopts different granularities in order to mitigate the overhead of data movement due to coherence invalidations. Our approach, on the other hand, addresses adaptivity when applied to the classification of data into private and shared.

Finally, POPS [Hossain et al. 2011] presents a protocol optimized for private or shared data to meet the trade-off between a private and a shared LLC, which treats migratory data as private.

3. GENERATIONAL CLASSIFICATION

We describe the generational classification at cache line granularity, however the method can be generalized to any granularity. We assume a system comprising of private L1 caches and a shared LLC. All L1s send their data requests to the LLC. Since LLC observes all the requests in the system, our classification mechanism is therefore located in the LLC. Classification is sent to L1s along with data responses from the LLC.

A generation for a cache line starts when the cache line is brought into an L1 cache as a result of an L1 miss. A cache line can therefore have many concurrent generations in different cores, up to a generation per core. A core may repeatedly access a cache line in its L1 cache before the cache line enters a *dead* time awaiting eviction. A generation terminates when the cache line is evicted from the cache as a result of its non-use. We use replacements as an approximation for the termination of generations, noting that when a line is replaced it might still be in active use—in its *live* time—but is evicted as result of a conflict in a cache with limited associativity. However, termination of a generation can take alternative interpretations. Termination can be explicitly signaled, for example, by a program instruction, or by a dead-block detection/prediction mechanism such as cache decay [Kaxiras et al. 2001; Lai et al. 2001] or more sophisticated dead-block predictors [Lai et al. 2001]. This gives us a means of controlling the generation length by modulating its dead time. We use this feature in the evaluation section to demonstrate this ability by using cache decay as a crude but simple predictor. We can also rely on other events, such as coherence invalidations, to determine the termination of a cache line generation. However, this gives a weaker definition for the concept of generation, as the cache line being invalidated might in fact be in active use, which is going to be requested again by the same core in a short time. Migratory data is a typical example for this case. As a result, we do not end the generations upon coherence invalidations. This is further discussed at the end of section 4.1.

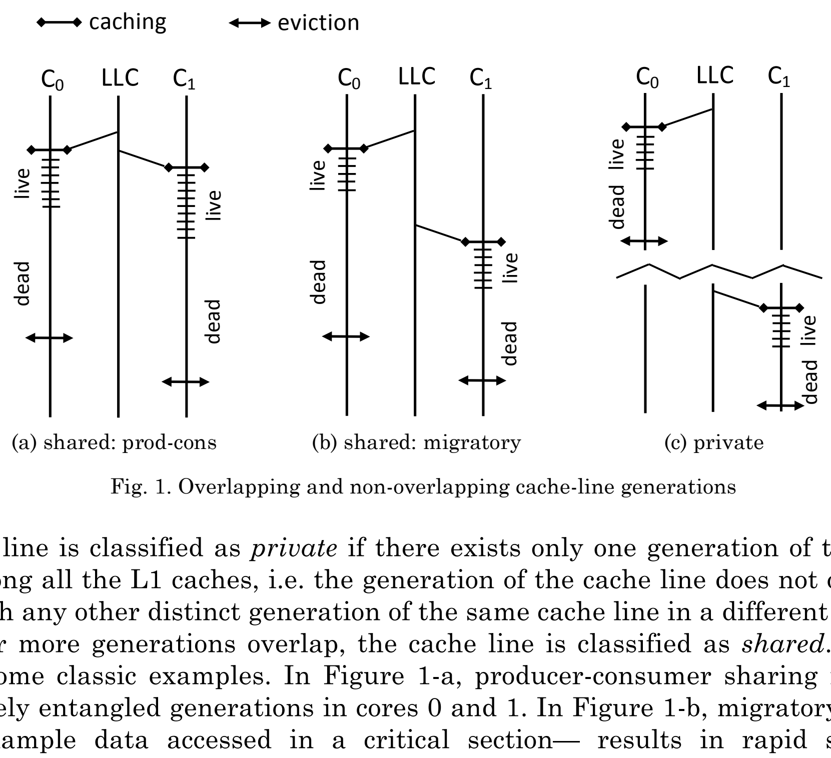


Fig. 1. Overlapping and non-overlapping cache-line generations

A cache line is classified as *private* if there exists only one generation of that cache line among all the L1 caches, i.e. the generation of the cache line does not overlap in time with any other distinct generation of the same cache line in a different L1 cache. If two or more generations overlap, the cache line is classified as *shared*. Figure 1 shows some classic examples. In Figure 1-a, producer-consumer sharing results in two closely entangled generations in cores 0 and 1. In Figure 1-b, migratory sharing-—for example data accessed in a critical section— results in rapid successive generations on different cores that tend to overlap in their dead times. In Figure 1-c, two generations of the same cache line exist on different cores but are sufficiently separated in time so as to consider the line essentially private. This is a common occurrence in pipelined parallelism where one core operates on some data for some time and then passes them to another core. In our approach, the differentiation of the last two cases focuses on the time separation of the generations. We do not explicitly identify migratory sharing, which can be done orthogonally to our approach [Stenström et al. 1993; Cox and Fowler 1993; Kaxiras and Goodman 1999]; we leave this for future work.

To track a generation of a cache line, the beginning and the end of the generation should be made visible to the classification mechanism. The beginning of a generation occurs as the result of an L1 miss, and is marked by a request to the LLC for the cache line (GET). Detecting the end of a generation, based on our chosen model described earlier, is heavily based on the underlying coherence protocol. Traditional directory-based coherence protocols, such as MSI (and all its derivatives), require notifications for all cache line replacements in the form of a PUT message when doing write-back or an explicit eviction notification (EEN) for clean lines [Sorin et al. 2011]. For such coherence protocols, each PUT and EEN message received at LLC could be used to mark the end of one generation for that cache line.

Data classification information is stored in the LLC tags using a *Private/Shared* bit and a dual purpose *PrivateOwner/SharerCount* field that holds the Private Owner ID in case of private classification, or the number of sharers for a cache line if the cache line is classified as shared. *PrivateOwner/SharerCount* field is of length $\lceil \log_2 n \rceil$, where *n* denotes the number of cores in the system. The following convention holds for the generational classification, where the first entry represents the *Private/Shared* status and the second entry represents *PrivateOwner/SharerCount*:

<Shared, 0>: NULL entry (not present in any L1 cache)
<Private, X>: Private, X is owner, X in {0 .. N-1} for N cores
<Shared, n>: Shared, n sharers, where $1 \le n \le N$

The classification entries are initialized to <Shared, 0>. Upon observing the first L1 request for a cache line in the LLC, *Private/Shared* bit is set to Private, and the second field is set to the ID of the requesting core. If GET_X denotes a GET request by core X, then the classification is formulated as:

$$GET_X : \langle Shared,\ 0\rangle \rightarrow \langle Private,\ X\rangle$$

For a further access to a cache line classified as private, classification will change to shared. The second field of attribute bits will now contain the number of sharers for that cache line:

$$GET_Y : \langle Private,\ X\rangle \rightarrow \langle Shared,\ 2\rangle$$

Upon receiving a replacement notification in the form of write-back for a cache line classified as private, classification will change to NULL for that cache line:

$$WB_X : \langle Private,\ X\rangle \rightarrow \langle Shared,\ 0\rangle$$

Upon receiving GET requests in the LLC for a cache line, which is already classified as shared, classification remains as shared, however, the *SharerCount* field will be updated to reflect the degree of sharing for that cache line:

$$GET_X : \langle Shared, n \rangle \rightarrow \langle Shared, n+1 \rangle$$

For any write-back or coherence INV-ACK message received at the LLC for a cache line classified as shared, degree of sharing is updated as follows:

$$INV\text{-}ACK \mid WB : \langle Shared, n \rangle \rightarrow \langle Shared, n-1 \rangle$$

A shared cache line reverts back to *private* state if all the overlapping generations of that cache line end. In our implementation, such a reverse classification adaptation can happen after the LLC has classified a cache line as <*Shared, 0*>. Upon creation of the next generation, the new generation will be born as private. Classification state of <*Shared, 1*> in essence denotes a private state. However, we cannot adapt the classification back to private at this point, since the owner is unknown. Therefore the classification mechanism we introduced so far will only allow classification adaptation from shared to private at a point that all the generations of a cache line have ended.

So far we introduced the concept of data classification based on generations of cache lines. We showed that such data classification works with the traditional directory-based coherence protocols, and it provides reverse data classification adaptation from shared to private. In the next section, we introduce our new coherence protocol, called *Generational Coherence*.

4. GENERATIONAL COHERENCE

Our new coherence protocol, which we call *Generational Coherence*, belongs to the class of coherence protocols that rely on software's data-race-free semantics and are based on self-invalidation [Lebeck and Wood 1995; Choi et al. 2011; Kaxiras and Ros 2012; Ros and Kaxiras 2012; Kaxiras and Ros 2013]. We motivate our choice from several different aspects: coherence protocols based on self-invalidation are simple to design, modify, and debug, therefore it is easier to study the impact of different optimizations for such protocols. Those protocols also reduce network traffic by eliminating the invalidation traffic, and improve the overall performance. Similar to VIPS-M, GC uses a dynamic write policy: write-back for modified private cache lines and delayed write-through for modified shared ones. Clean private cache lines are replaced silently from L1 caches. In our protocol, each core silently self-invalidates all its shared data. The generational data classification discussed in the previous section should therefore be adjusted to accommodate such write policy and also the side effects of silent self-invalidations.

Self-invalidations are silent, invisible to LLC, and do not affect the termination of generations. We choose not to require explicit self-invalidation notifications to the LLC, since such requirement would be contrary to what coherence protocols based on self-invalidation are trying to achieve. A core locally self-invalidates its cache line by setting the *invalid* bit for the cache line. Subsequent cache lookups can detect if a cache line is self-invalidated when the tag is found in the cache and the corresponding *invalid* bit is set. This allows a core to detect if a cache miss is due to self-invalidation —coherence miss— or belongs to any of the 3C cache-miss types [Hill and Smith 1989]. To allow our generational classification to tolerate the aforementioned silent self-invalidation, GET requests following the cache misses that are due to self-invalidation —coherence miss— should be distinguished from GET requests following the cache misses which belong to other cache-miss types: any subsequent cache miss which is not caused by an earlier self-invalidation issues a normal GET request to the LLC to obtain the cache line *and signal to the classification mechanism the start of a generation.* Cache misses that are caused by earlier self-invalidations, however, are treated differently since they happen in the middle of an existing generation which is already accounted for in the classification. Self-invalidated cache lines are therefore required to be *re-fetched* from the LLC using a special GET message, hereafter called REFRESH.

Upon receiving a REFRESH message, the LLC does not create a new generation for that cache line, and the number of sharers is not incremented. Classification in that case is formulated as:

$$REFRESH : \langle Shared,\ n\rangle \rightarrow \langle Shared,\ n\rangle$$

Using REFRESH messages allows yet another optimization: the time frame for shared-to-private adaptation can be shrunk by one step. We mentioned earlier that shared-to-private adaptation is only possible when all the generations of a cache line have ended, and a new generation is about to be created from the NULL state, i.e. the reverse classification adaptation takes place in state <*Shared, 0*>*,* and not <*Shared, 1*>. This is due to the fact that the owner of a cache line is unknown as soon as a cache line is classified as shared. With REFRESH requests, however, it is possible to adapt the classification back to private from state <*Shared, 1*> immediately. Since <*Shared, 1*> state denotes that there exists only one sharer/generation of the cache line among all the private caches, a REFRESH request reveals the identity of the core that holds the only generation of that cache line. At this point, the classification mechanism can safely perform the classification adaptation, and the cache line will be classified as private.

Such optimization is also possible when a write-through is observed at LLC for a cache line in state <*Shared, 1*>. Similar to a REFRESH, a write-through in state <*Shared, 1*> reveals the only owner of that cache line, and adaptation is triggered:

$$REFRESH_X \mid WT_X : \langle Shared,\ 1\rangle \rightarrow \langle Private,\ X\rangle$$

In order to be able to signal the termination of generations, we require that cache lines classified as shared explicitly notify LLC of their eviction. This also applies to the self-invalidated cache lines. Clean-shared cache lines and self-invalidated cache lines notify their eviction via EEN, while modified shared cache lines are written back to LLC:

$$EEN \mid WB : (Shared,\ n) \rightarrow (Shared,\ n\text{-}1)$$

Private blocks that contain modified data will be written back into LLC, and therefore their generation will end. For the clean private cache lines, however, we do not require EEN. Since there are private blocks that remain private to a single core for the whole program execution, it is therefore more efficient not to track the end of the generations of clean private blocks via EENs. Clean private blocks are therefore replaced silently. Our generational coherence mechanism detects such replaced cache lines after receiving new requests for those cache lines using a mechanism called *Recovery*.

4.1 Recovery

For an access to a cache line already classified as private, the *PrivateOwner* field of the cache line is compared to the ID of the core that initiated the request. If the *PrivateOwner* field matches the ID of the requesting core, the cache line remains private with the same private-owner.

$$GET_X : (Private, X) \rightarrow (Private, X)$$

This is the case where a private-clean cache line is silently evicted due to capacity/conflict misses, and requested again by the previous private owner.

If the two IDs are different, the former private owner must be notified by a request from LLC, before LLC can respond to the new request. This recovery notification happens only once per private-to-shared transition, and is in form of a unicast, since only the private owner needs to be notified. Two scenarios are possible: if the former private owner has the line in its cache, it changes the classification of the line in its cache from private to shared. As a result of changing classification, the former owner either performs a write-back of dirty data, or sends an acknowledgement in case of a clean private block. The cache line classification then changes to Shared in the LLC, and the *PrivateOwner/SharerCount* field is set to 2, denoting the number of sharers:

$$GET_Y : (Private, X) \rightarrow (Shared, 2)$$

However, if the former private owner has silently evicted the line from its cache—i.e. the cache line generation has ended—, the former private owner replies with a negative acknowledgement (NACK) to the LLC request. In this case, the cache line classification remains private in the LLC, and the *PrivateOwner/SharerCount* field is set to the ID of the new owner:

$$GET_Y : (Private, X) \rightarrow (Private, Y)$$

Before LLC can respond with data to the new request *Y* in the aforementioned scenario, LLC needs to access the memory if data is also evicted from the LLC.

Backward adaptation, if not performed with care, can result in performance degradation due to excess recovery overhead caused by repetitive and useless private-to-shared and shared-to-private transitions. This is the typical case when migratory data is prematurely classified as private [Pugsley et al. 2010]. This happens if self-invalidations are used to signify the end of generations. To fend against this, we consider that generations only end when cache-lines are evicted, not when they are invalidated. We ensure that their classification remains shared through synchronization points, causing no additional recoveries. If migratory data is misclassified as private, the recovery process adds latency to the migration, whereas shared modified migratory blocks are written-through to the LLC prior to synchronization, an action whose latency is overlapped and hidden by other delayed write-throughs.

4.2 Read-Only Recovery

As a further optimization to mitigate the penalties associated with self-invalidation, we enrich the data classification by considering read-only (RO) classification. Similar to private data, the shared data classified as RO is also spared from self-invalidation, which yields higher hit rate. A cache line is classified as read-only as long as no *write* is observed in the LLC for that cache line. Upon observing the first *write* in the LLC, which is either in the form of a write-through or a GETX —request with *write* permission— the cores that have the cache line are notified by the LLC to update the classification for that cache line in their L1 caches from read-only to read-write. Such operation, which we refer to as *RO-Recovery*, is not considered to be costly, since it happens only once per read-only to read-write transition. Furthermore, *RO-Recovery* is not on the critical path. Upon receiving a *RO-Recovery* notification, cores do not need to be stalled in order to perform the recovery immediately. It is sufficient that the cores perform *RO-Recovery* before their next synchronization.

4.3 Loss of classification information

The Private/Shared classification and private owner field exist only for the LLC lines. The status bit and *PrivateOwner* field are not saved externally, and are lost upon eviction of the cache line from the LLC. When a cache line is initially brought into LLC, the state of the *Private/Shared* bit can vary, depending upon whether the cache hierarchy is inclusive or non-inclusive.

For an inclusive hierarchy, when a cache line is evicted from LLC its L1 copies must also be evicted. There are three cases depending on the classification state of the LLC line. In the first case, the line is in state NULL and there are no sharers. This is a common case since the eviction from LLC indicates that line has not seen much activity for a long period. In the second case, the line is private and a recovery notification to the known private owner is sent to evict it. In the last case, the line is shared and only the number of sharers is known. This is more expensive since it requires a broadcast. To minimize its effects, preference may be given to the other cases over this in the replacement algorithm. The LLC eviction is not on the critical path of the miss that causes the eviction. LLC eviction is handled by a MSHR. When a new cache line is brought into the LLC, the requesting L1 cache becomes the line's private owner and the cache ID is added to the private owner field.

Alternatively, for a non-inclusive hierarchy, when a line is brought into LLC as a result of an L1 miss, the classification is unknown and must be reconstructed by querying the L1s. A broadcast—snoop—to all the L1s establishes which (if any) L1 has the line. If an L1 has the line, it replies with its ID, otherwise with a NACK. If more than one L1 has the line, then the line is shared and *SharerCount* is calculated based on the received ACK messages. If only one L1 has the line, its ID becomes the *PrivateOwner* value. If no L1 has the line—i.e. an LLC cold miss—, the ID of the requesting L1 cache is used as private owner. Once the Private/Shared status is established, the classification is performed anew for the requesting L1. Broadcasts in the L1s concern only LLC misses, which are significantly fewer than L1 misses. Furthermore, reconstructing the classification is also not in the critical path of the LLC miss since it is overlapped with memory access.

Table I. Comparison of Protocol States

	GC		**MESI**	
	Stable	Transient	Stable	Transient
L1	4	4	6	8
LLC	2	2	4	5

4.4 Protocol complexity and protocol races

Adding LLC classification to a directory-less protocol such as VIPS-M may seem to increase complexity by re-introducing directory functionality. Without a doubt, GC is more complex than VIPS-M but the complexity gap to a directory protocol such as MESI is still substantial. This is clearly evident in the number of protocol states for GC and MESI, shown in table I. Classification can merely be thought of as a passive storage that simply observes the requests that reach the LLC, and via a simple FSM updates a state variable. Classification does not entail request forwarding or explicit coherence invalidations, which are responsible for the complexity in directory protocols. The only request sent on the part of classification is the *recovery notification*—and similarly, the recovery notifications when we need to reconstruct the classification information from scratch. However, this does not add any complexity to VIPS-M that was not already there. Recovery is an essential operation in the original protocol whether it is done at page level or line level: any transition from private to shared must notify the private owner so the page or cache line becomes shared in the L1.

5. EVALUATION METHODOLOGY

We evaluate GC against a directory protocol (MESI states) and also a self-invalidation request-response protocol which performs private/shared data classification at page granularity (VIPS-M), in order to study the impact of generational data classification on (i) the amount of shared/private data classification, (ii) network traffic, and (iii) execution time. We implement two versions of GC: a version without any dead-block prediction mechanism, and another that employs cache decay as a simple predictor [Kaxiras et al. 2001]. We use the Simics full-system simulator [Magnusson et al. 2002], and model VIPS-M and GC protocols using the cycle-accurate GEMS simulator [Martin et al. 2005]. We also employ the GARNET network simulator [Agarwal et al. 2009] to model the interconnection network. Our target system is a 16-tile chip multiprocessor. Table II gives details about the main parameters of our base system. Furthermore, we use Pin [Luk et al. 2005] in order to study and analyze the shared data access patterns.

We employ a wide variety of parallel applications. *Barnes* (16K particles), *Cholesky* (tk16), *FFT* (64K complex doubles), *FMM* (16K particles), *LU-CB* (512x512 matrix), *LU-NCB* (512x512 matrix), *Ocean* (514x514 ocean, contiguous partitions), *Radiosity* (room, -ae 5000.0 -en 0.050 -bf 0.10), *Raytrace* (teapot, optimized version that removes unnecessary locks), *Volrend* (head), *Water-Nsq* (512 molecules) and *Water-Sp* (512 molecules) belong to the *SPLASH-2* benchmark suite [Woo et al. 1995]. *Blackscholes* (simsmall), *Canneal* (simsmall), and *Swaptions* (simsmall) are from the PARSEC benchmark suite [Bienia et al. 2008]. We simulate the entire applications, but collect statistics only from start to completion of their parallel part.

Table II. Base System Parameters

Memory Parameters	
Processor frequency	3.0 GHz
Block size	64 bytes
MSHR size	16 entries
Split L1 I & D caches	32 KB, 4-way
L1 cache hit time	1 (tag) and 2 (tag + data) cycles
Shared unified LLC cache	8 MB, 512 KB/tile, 16-way
LLC bank cache hit time	6 (tag) and 12 (tag + data) cycles
L1-LLC inclusion policy	Inclusive
MESI Directory	Full-map in LLC tags
Memory access time	160 cycles
Page size	4 KB (64 blocks)
Network Parameters	
Topology	2-dimensional mesh (4x4)
Routing technique	Deterministic X-Y
Flit size	16 bytes
Data message size	72 bytes (5 flits)
Control message size	8 bytes (1 flit)
Routing time	2 cycles
Switch time	2 cycles
Link time	2 cycles

6. RESULTS

6.1 Classification Quality

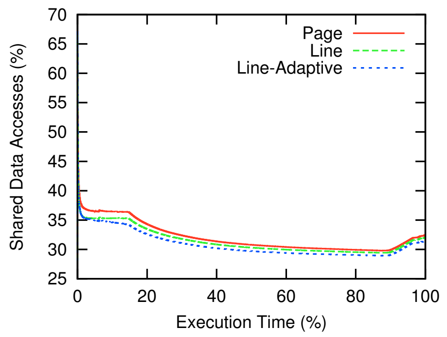
6.1.1. Granularity and Adaptivity Do Not Matter. Figure 2 shows the benchmarks that do well with page-level classification. Granularity or adaptivity has no significant effects on the outcome of classification, as the number of accesses to the data classified as shared remains almost the same for both granularities. This is the case where most of the cache lines in a page classified as shared are truly shared. *Blackscholes*, which is not included in the figure due to space limitations, also shows a trend very similar to *Water-SP*.

6.1.2. Granularity Matters. Figure 3 shows the benchmarks that benefit from fine-grained classification. Such benchmarks have shared pages with the majority of cache lines in those pages being private. Fine-grained classification prevents such private cache lines from being misclassified as shared.

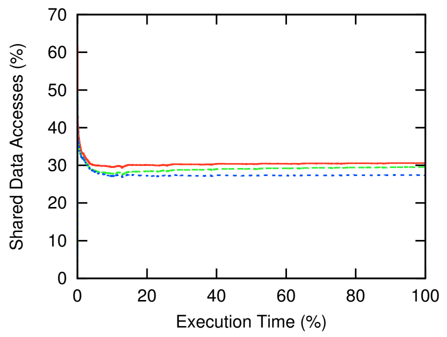
6.1.3. Granularity and Adaptivity Matter. Benchmarks in Figure 4 benefit from both fine-grained classification and adaptation. Adaptation serves to prevent temporarily private data from being misclassified as shared [Alisafaee12].

6.2 Effects on Performance

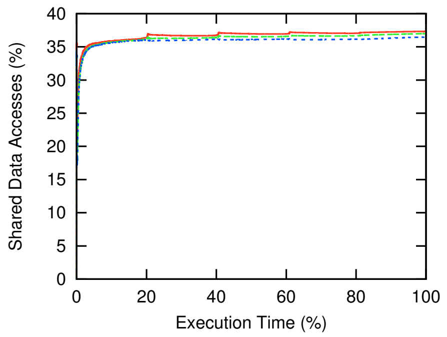
In this section we compare page-based non-adaptive coherence (VIPS-M) and block-based, adaptive coherence (GC). GC aims to further enhance VIPS-M by having less shared data. We also compare miss rate, network traffic, and execution time against a MESI protocol to provide a general comparison against invalidation-based coherence protocols.



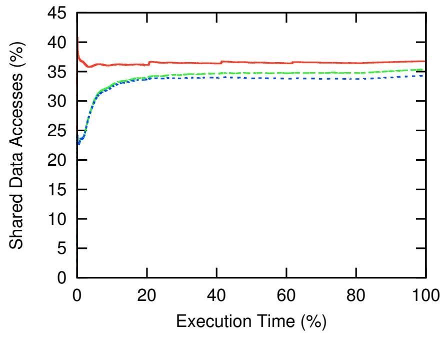
(a) SPLASH-2/Radiosity



(b) SPLASH-2/Volrend

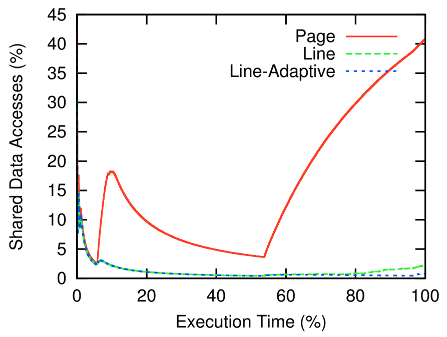


(c) SPLASH-2/Water-nsq

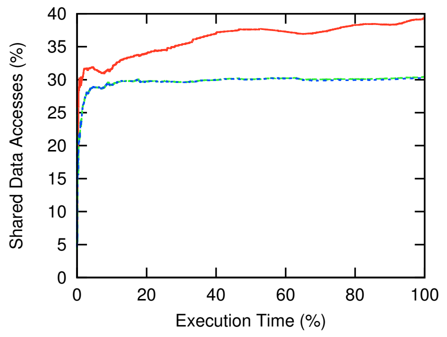


(d) SPLASH-2/Water-sp

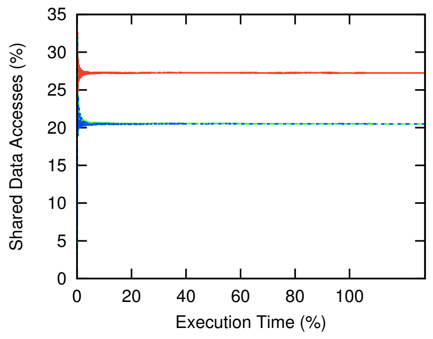
Fig. 2. Benchmarks that are insensitive to granularity/adaptivity (run-time average)



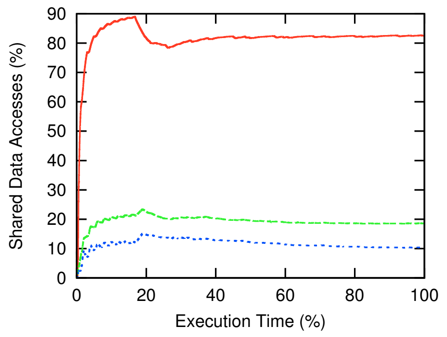
(a) SPLASH-2/FFT



(b) SPLASH-2/LU-CB

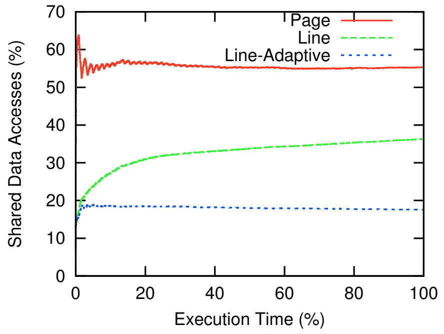


(c) PARSEC/Swaptions

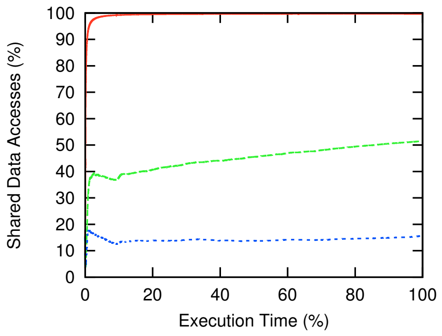


(d) SPLASH-2/Ocean-CP

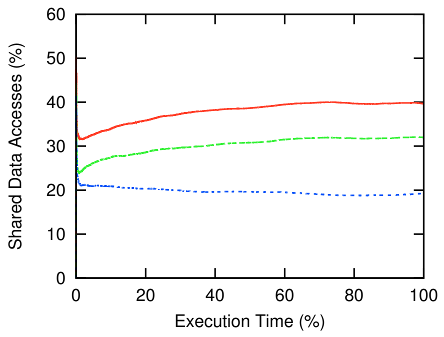
Fig. 3. Benchmarks sensitive to granularity only (run-time average)



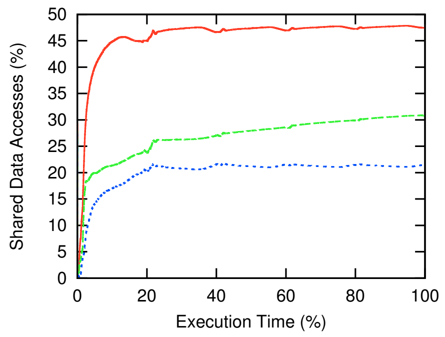
(a) PARSEC/Canneal



(b) SPLASH-2/LU-NCB



(c) SPLASH-2/Raytrace



(d) SPLASH-2/FMM

Fig. 4. Benchmarks sensitive to both granularity and adaptivity (run-time average)

Figure 5 shows the amount of data classified as private and shared in the L1 caches of a 16-core multicore processor. *Canneal* appears as mostly private in the graph due to having locks mainly outside region of interest and having significantly low amount of shared data. The graph also includes the portion of shared data classified as read-only (RO). VIPS-M obtains RO information from page table entries provided by OS, while GC keeps track of RO blocks dynamically as discussed in section 4.2.

As depicted in Figure 5, GC classifies on average about three times fewer shared-written data and about three times more private data compared to VIPS-M. This illustrates the significant impact of fine-grained adaptive data classification on the amount of data classified as private.

We also employ cache decay [Kaxiras et al. 2001] to modulate the dead time of the cache-line generations. We consider only a simple dead-block predictor using a two-bit saturating counter per cache line in L1 caches, since sophisticated predictors would incur complexity and cost disproportionate to the rest of our mechanisms. We apply decay selectively to private data, since the generations of shared data seem to operate on markedly different time scales. Contrary to the work in [Kaxiras et al. 2001], we do not turn off cache lines that are believed to have entered their dead time. Our intent is simply to understand the effect of cutting short a generation once we have established that it has entered its dead time. Thus, in our approach, decay is simply a marker in the timeline of a generation, after which the cache line can remain private if requested by another core. A recovery request that finds a line after its decay, forces its eviction; the line can be passed on to the requesting core as private. In contrast, if the original private owner accesses the cache line after it has decayed (without an intervening recovery), the line is *revived* and reinstated as live, without causing a decay miss.

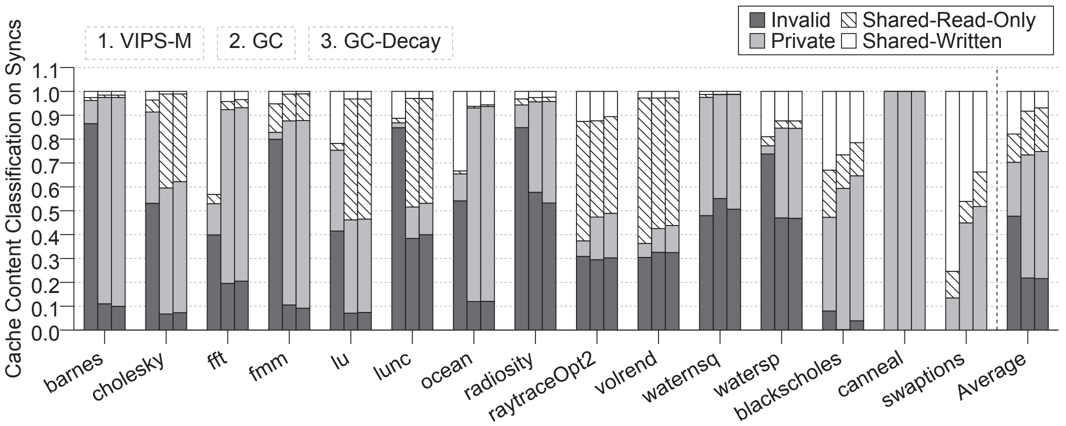


Fig. 5. Classification quality VIPS-M vs. generational data classification

Given this flexibility we can set fairly small decay intervals without fear of generating significant decay misses. We use decay intervals of ranging from 500 cycles to 40K cycles. Our results show that attempting to shorten the dead time has positive but small effects. With a very small decay interval—e.g. aggressive dead-block prediction—we run the danger of trying to classify as private, data that are inherently shared. This leads to increased recoveries (re-classifying as shared what we artificially classified as private), thus eliminating any potential benefit. This leaves medium to large decay intervals as the best performing (e.g., 10K to 20K cycles), but at that point dead-block prediction does not differ much from actual eviction as far as the classification is concerned. For this reason, attempting to control the length of generations is likely not to be a fruitful direction. We include the results using decay for completeness.

As Figure 5 shows, VIPS-M has a larger portion of its caches as invalid. This is due to the fact that VIPS-M has more shared data, which are self-invalidated upon synchronizations. GC, on the other hand, classifies more data as private, which are not affected by synchronizations.

Figure 6 shows that GC slightly reduces the miss rate compared to VIPS-M. Although one would expect substantial decrease in miss rate due to the sharp increase in the amount of private data, the reduction is marginal. We explain this in the next section, where we show the rate of re-accessing self-invalidated data. There are cases where using cache decay slightly degrades the miss rate. This is due to mispredicting entry into the dead time and, because of an intervening recovery, the generation is ended too early.

GC incurs some overhead. Such overhead is due to *recovery* and *EEN* messages needed to track beginning and end of generations. Despite such overhead, reduction in overall network traffic and energy consumption is still possible. GC reduces the network traffic up to 30% in *Watersp*, and about 20% in some benchmarks shown in Figure 7. One might wonder where the reduction in the network traffic comes from, as Figure 6 does not show significant drop in the miss rate. We discuss this in the next section where we show the impact of granularity and adaptation on the amount of write-through traffic. Finally, Figure 8 shows that despite incurring the granularity and adaptation overhead, GC does not degrade the execution time compared to VIPS-M, while reducing the overall network traffic and consequently the energy consumption.

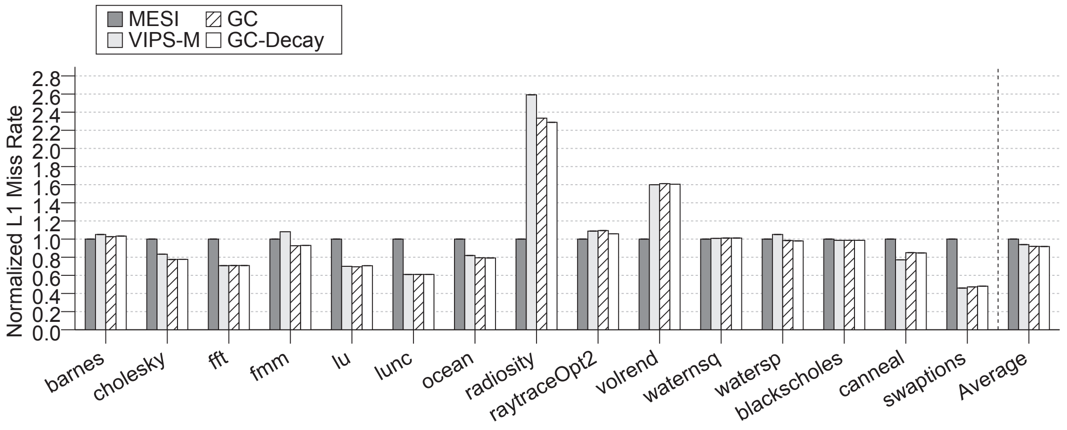


Fig. 6. L1 cache miss rate

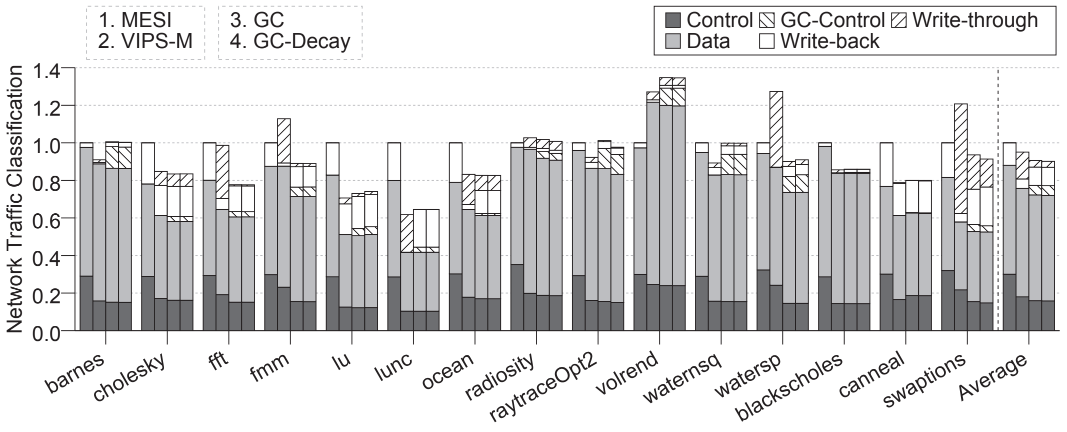


Fig. 7. Network traffic

6.3 Explaining the Results

6.3.1. The impact of granularity and adaptation on miss rate. One would expect a lower miss rate for coherence protocols based on self-invalidation when the amount of private data significantly overweighs the amount of shared data. However, as shown in Figure 6, GC only slightly decreases the miss rate. This can be explained by referring to Figure 9. Almost all of the benchmarks have similar low rate of re-accessing the self-invalidated data, regardless of granularity and adaptation. The benchmarks that are not shown in Figure 9 have re-access rate close to zero. Since the self-invalidated data is not re-accessed, it does not matter if the data are classified as private. A slightly more decreased miss rate for some of the benchmarks such as *Radiosity*, *FMM*, *Water-SP*, and *Cholesky* can be explained by referring to synchronizations. Such benchmarks have a large number of locks and spend significant portion of their execution time (30% in *Radiosity*) in synchronization.

Frequent synchronizations result in frequent self-invalidation of all shared data. Thus, even small changes in the re-access rate are significantly amplified. As Figure 6 shows, GC decreases the miss rate for *Radiosity* about 10%.

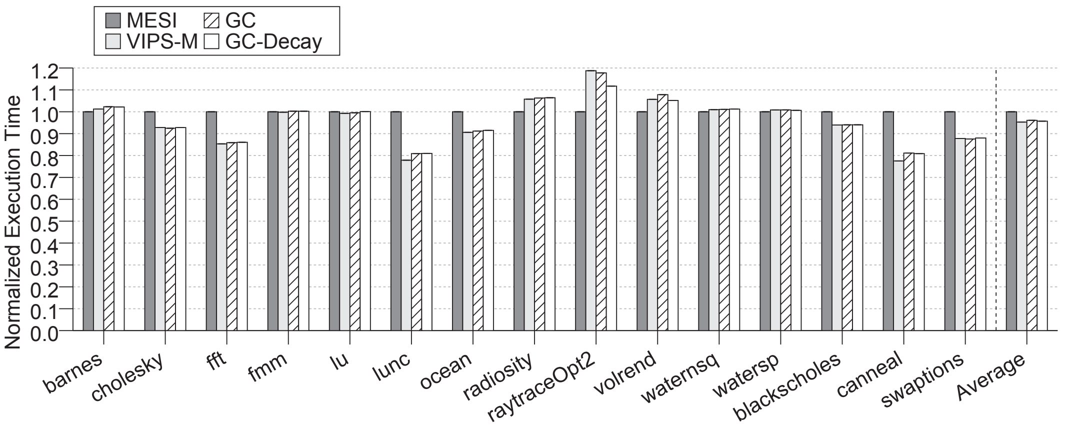


Fig. 8. Execution time

Figure 6 shows that GC results in a slightly higher miss rate for benchmarks such as *Canneal* and *Swaptions*. This is mainly due to the self-invalidated cache lines being deallocated from the cache in VIPS-M. However, in GC the generations of cache lines do not end by self-invalidations, causing more private blocks to be victimized by the replacement algorithm. The penalty is negligible, and can be fixed by giving priority to self-invalidated blocks in the replacement algorithm.

6.3.2. The impact of granularity and adaptation on network traffic. The reduction in network traffic without significant decrease in miss rate can be explained by referring to Figure 10, which shows the reduction in the amount of write-through traffic. Write-throughs have a major contribution to network traffic in protocols that employ write-through policy for shared data. Therefore, finer granularity and adaptation will result in network traffic reduction by reducing the amount of shared data. Benchmarks such as *FFT*, *FMM*, *Water-SP*, and *Swaptions*—*Swaptions* in not included in Figure 10 due to space limitations—significantly benefit from finer granularity and adaptation, as the amount of write-through traffic is significantly reduced for these benchmarks. There are also benchmarks, such as *LU-NCB*, *Ocean-CP* and *Blackscholes*, where network traffic is not decreased despite reduction in the amount of write-through—*Blackscholes*, which is not included in the figure due to space limitations, shows a trend very similar to *Water-SP*. Such behavior can be explained by referring to Figure 7. For those benchmarks, the amount of write-back has significantly increased, which cancels out the benefit of having less write-through traffic. This is the typical case for larger data sets, enforcing frequent replacements in L1 caches. *Canneal* has very low amount of shared data that makes it insensitive to write-throughs, behaving almost the same with both GC and VIPS-M. It has slightly higher network traffic with GC due to slightly higher miss rate.

There are also benchmarks in Figure 7 such as *Barnes, Raytrace, Volrend, and Water-nsq*, where the increase in network traffic incurred by finer granularity and adaptation is more pronounced. These are the benchmarks that have almost the same amount of write-through and write-back traffic regardless of granularity and adaptation. However, the overhead of maintaining a fine-grained classification increases the total network traffic for those benchmarks. Although GC has lower write-through rate than VIPS-M for those benchmarks, the amount of shared data is considerably low, making the difference in write-through traffic negligible.

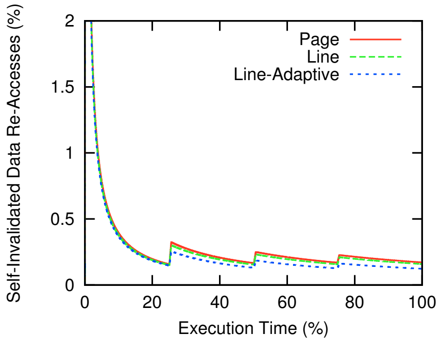
Up to this point we referred to write-through rate to explain the reduction in network traffic despite having similar miss rate with GC and VIPS-M. Although benchmarks that have lower miss rate with GC incur less data movement traffic due to cache misses, it is easy to observe that the impact of write-through is still dominant (Figure 6 and Figure 7). *Radiosity*, for example, which has the highest reduction in the miss rate with GC, shows no significant improvement in the network traffic despite having less data movement due to fewer cache misses. This can be explained by referring to Figure 7 and Figure 10, which reveals that *Radiosity* has almost the same amount of write-through traffic regardless of granularity and adaptation: finer granularity or adaptation are unable to significantly reduce the amount of shared data for this benchmark (Figure 5). In general, benchmarks are more sensitive to write-through traffic rather than data traffic caused by cache misses. The same holds for *Cholesky*. *FMM* and *Water-SP* also have lower miss rates with GC, however their reduction in network traffic does not come from the reduced data movement due to lower miss rate, but from significant reduction in the amount of write-through.

6.3.3. Putting it all together. Our results show that:

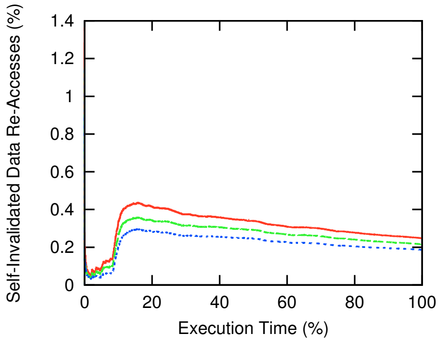
- For many benchmarks adaptive block-level classification significantly reduces the amount of shared data, yet does not affect the overall miss rate (Figure 6).

- The data re-classified as private using adaptive block-level classification, which would have otherwise been classified as shared at page granularity, in many cases, cause a noticeable reduction of write-throughs and network traffic (Figure 7) and consequently result in lower energy consumption.

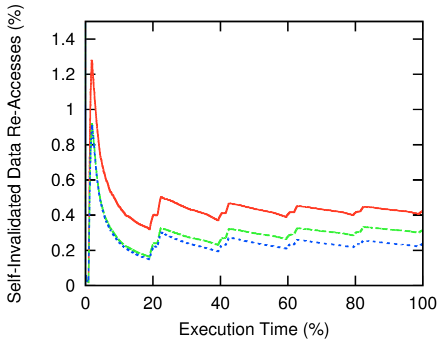
How can these two seemingly contradictory observations be reconciled? The answer lies in the dynamics of the generational behavior upon which our classification is based. First, most of the data re-classified as private are dead before synchronization and are not re-accessed immediately after. In other words, the live time of a single generation of such data does not typically span across synchronization points. This is evidenced by the generally small re-access rate of self-invalidated data (Figure 9), which means that the behavior of the miss rate is dominated by capacity misses. Any change in self-invalidation misses is hardly noticeable ---except when magnified by very frequent synchronization as in the case of *Radiosity*. However, while most of the data, re-classified by generational coherence as private, are dead at synchronization points, this does not mean that they are not re-accessed again at a much later time, starting a new generation. The compound effect of all such private generations is to reduce write-through traffic. When this effect is not balanced out by write-back traffic or control overhead, the overall network traffic is reduced, which in turn results in lower energy consumption.



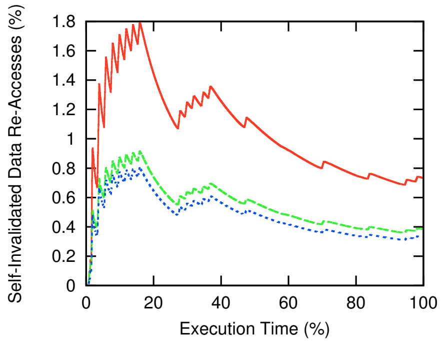
(a) SPLASH-2/Barnes



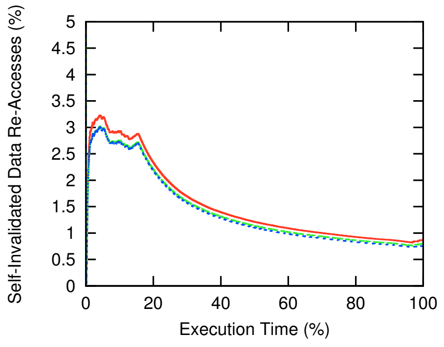
(b) SPLASH-2/Cholesky



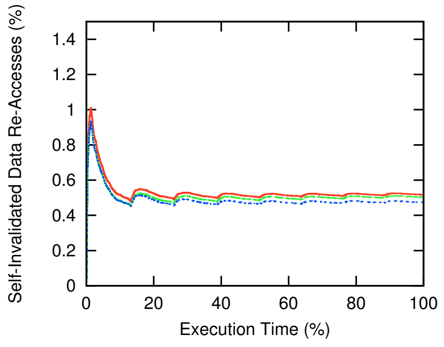
(c) SPLASH-2/FMM



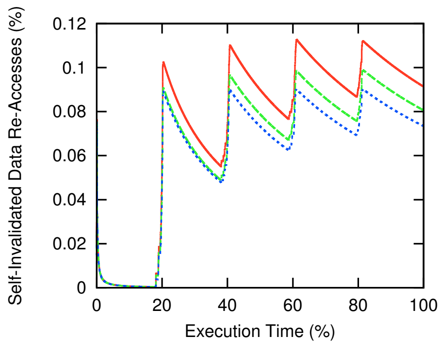
(d) SPLASH-2/Ocean-CP



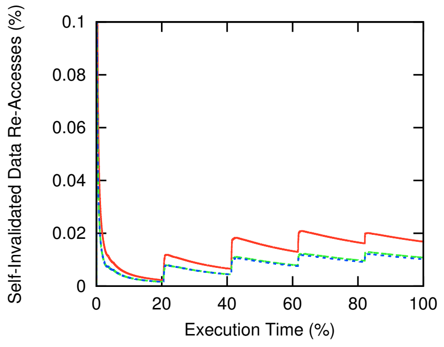
(e) SPLASH-2/Radiosity



(f) SPLASH-2/Volrend

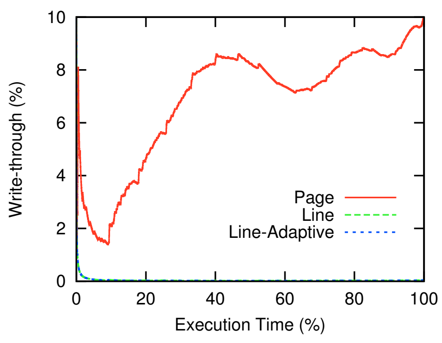


(g) SPLASH-2/Water-NSQ

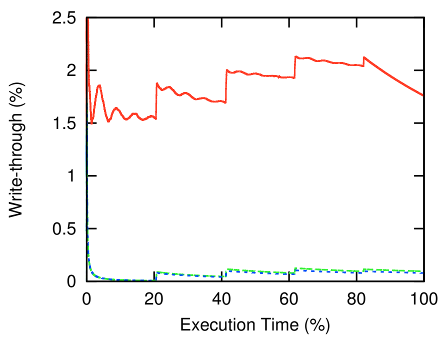


(h) SPLASH-2/Water-SP

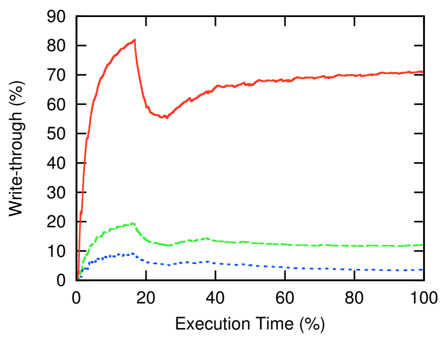
Fig. 9. Self-Invalidated data re-access rate (run-time average)



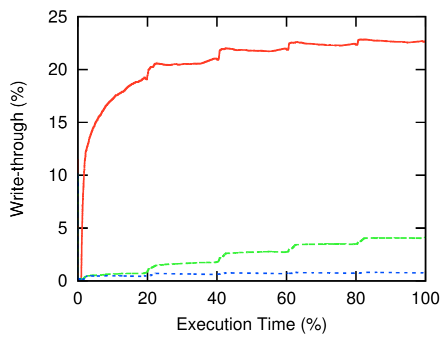
(a) SPLASH-2/LU-CB



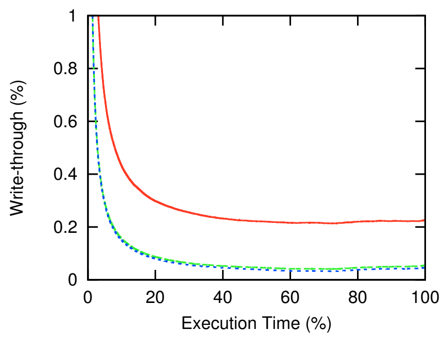
(b) SPLASH-2/Water-SP



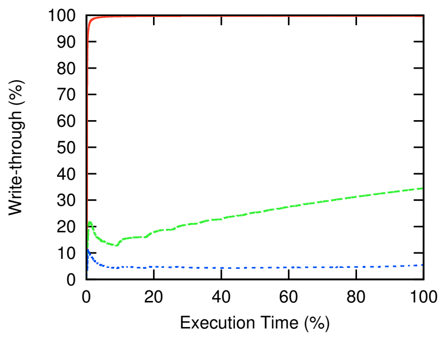
(c) SPLASH-2/Ocean-CP



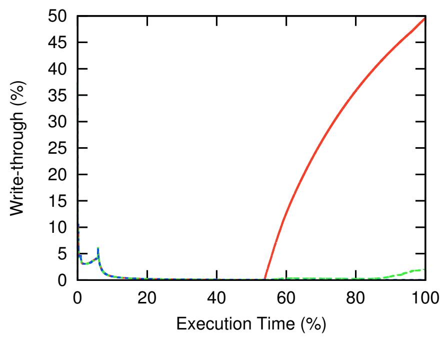
(d) SPLASH-2/FMM



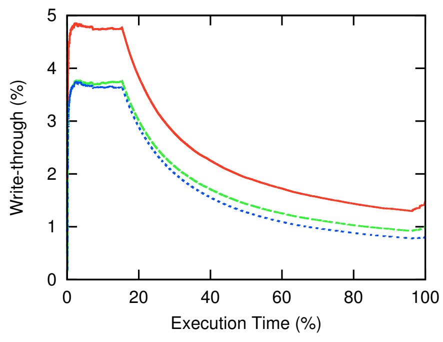
(e) SPLASH-2/Raytrace



(f) SPLASH-2/LU-NCB



(g) SPLASH-2/FFT



(h) SPLASH-2/Radiosity

Fig. 10. Granularity/adaptivity impact on write-through traffic (run-time average)

7. CONCLUSIONS

Private/shared data classification has become an essential part of many approaches to optimize cache coherence. Such data classification can be performed at page or cache-line granularity, with or without adaptation from shared to private. Regardless of the used granularity, non-adaptive data classification suffers from a *shared* classification bias, in which eventually all the data in the system are classified as shared, degrading the intended coherence optimizations.

In this paper, which to the best of our knowledge is the first study of this kind, we studied the impact of finer granularity and adaptation on the quality of private/shared data classification and also on the performance of coherence protocols based on self-invalidation and write-through for shared data. To this end, we proposed a new data classification scheme and coherence protocol, which we call *generational classification* and *generational coherence* (GC), respectively. Our results show that benchmarks are less sensitive to fine-grained data classification in terms of miss rate, as the significantly large amount of data re-classified as private are not reused after synchronization, and the self-invalidated data re-access rate remains the same regardless of granularity and adaptation. The significantly more data classified as private by GC can result in lower network traffic by excluding them from write-through, thus yielding an overall benefit despite the control-traffic overhead, which in turn results in more energy-efficient systems.

REFERENCES

Sarita V. Adve and Mark D. Hill. 1990. Weak ordering – a new definition. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA),* Jun. 1990, 2–14.

Niket Agarwal, Tushar Krishna, Li-S. Peh, and Niraj K. Jha. 2009. GARNET: A detailed on-chip network model inside a full-system simulator. In *Proceedings of the IEEE Int'l Symposium on Performance Analysis of Systems and Software (ISPASS),* Apr. 2009, 33–42.

Mohammad Alisafaee. Spatiotemporal coherence tracking. 2012. In Proceedings of the 45th IEEE/ACM International Symposium on Microarchitecture (MICRO), Dec. 2012, 341–350.

Christian Bienia, Sanjeev Kumar, Jaswinder P. Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT),* Oct. 2008, 72–81.

Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-T. Chou. 2011. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. 2011. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT),* Oct. 2011, 155–166.

Alan L. Cox and Robert J. Fowler. 1993. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20st International Symposium on Computer Architecture (ISCA),* May 1993, 98–108.

Blas A. Cuesta, Alberto Ros, María E. Gómez, Antonio Robles, and José F. Duato. 2011. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA),* Jun. 2011, 93–103.

Lei Fang, Peng Liu, Qi Hu, Michael C. Huang, and Guofan Jiang. 2013. Building expressive, area-efficient coherence directories. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT),* Sep. 2013, 299–308.

Christian Fensch and Marcelo Cintra. 2008. An OS-based alternative to full hardware coherence on tiled CMPs. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA),* Feb. 2008, 355–366.

Michael Ferdman, Pejman Lotfi-Kamran, Ken Balet, and Babak Falsafi. 2011. Cuckoo directory: A scalable directory for many-core systems. In *Proceedings of the 17th International Symposium on High-Performance Computer Architecture (HPCA),* Feb. 2011, 169–180.

Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA),* Jun. 2009, 184–195.

M. D. Hill and A. J. Smith. 1989. Evaluating associativity in CPU caches. *IEEE Transactions on Computers (TC),* vol. 38, no. 12, (Dec. 1989) 1612–1630.

Hemayet Hossain, Sandhya Dwarkadas, and Michael C. Huang. 2011. POPS: Coherence protocol optimization for both private and shared data. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT),* Oct. 2011, 45–55.

Stefanos Kaxiras and James R. Goodman. 1999. Improving CC-NUMA performance using instruction-based prediction. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture (HPCA)*, Jan. 1999, 161–170.

Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. 2001. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA),* Jun. 2001, 240–251.

Stefanos Kaxiras and Alberto Ros. 2012. Efficient, snoopless, soc coherence. In *Proceedings of the 25th IEEE International System-on-Chip Conference (IEEE SOCC),* Sep. 2012, 230–235.

Stefanos Kaxiras and Alberto Ros. 2013. A new perspective for efficient virtual-cache coherence. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA),* Jun. 2013, 535–547.

Daehoon Kim, Jeongseob Ahn, Jaehong Kim, and Jaehyuk Huh. 2010. Subspace snooping: Filtering snoops with operating system support. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT),* Sep. 2010, 111–122.

An C. Lai, Cem Fide, and Babak Falsafi. 2001. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA),* Jun. 2001, 144–154.

Alvin R. Lebeck and David A. Wood. 1995. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA),* Jun. 1995, 48–59.

Yong Li, Ahmed Abousamra, Rami Melhem, and Alex K. Jones. 2010. Compiler-assisted data distribution for chip multiprocessors. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT),* Sep. 2010, 501–512.

Yong Li, Rami Melhem, and Alex K. Jones. 2012. Practically private: Enabling high performance CMPs through compiler-assisted data classification. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT),* Sep. 2012, 231–240.

Chi-K. Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay J. Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* Jun. 2005, 190–200.

Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *IEEE Computer*, (Feb. 2002), vol. 35, no. 2, 50–58.

Milo M. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. In *Computer Architecture News*, (Sep. 2005), vol. 33, no. 4, 92–99.

Seth H. Pugsley, Josef B. Spjut, David W. Nellans, and Rajeev Balasubramonian. 2010. SWEL: Hardware cache coherence protocols to map shared data onto shared caches. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT),* Sep. 2010, 465–476.

Alberto Ros and Stefanos Kaxiras. 2012. Complexity-effective multicore coherence. In Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT), Sep. 2012, 241–252.

Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. A Primer on Memory Consistency and Cache Coherence. In *Synthesis Lectures on Computer Architecture*, Mark D. Hill, Ed. Morgan & Claypool Publishers, 2011.

Per Stenström, Mats Brorsson, and Lars Sandberg. 1993. An adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20st International Symposium on Computer Architecture (ISCA),* May 1993, 109–118.

Stephen Somogyi. 2004. Memory coherence activity prediction in commercial workloads. Master of Science Degree Theses. Department of Electrical and Computer Engineering. Carnegie Mellon University. Pittsburgh, Pennsylvania.

Stephen Somogyi, Thomas F. Wenisch, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. 2004. Memory coherence activity prediction in commercial workloads. In *Proceedings of the Third Workshop on Memory Performance Issues (WMPI-2004).*

Steven C. Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder P. Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA),* Jun. 1995, 24–36.

David A. Wood, Mark D. Hill, and Richard E. Kessler. 1991. A model for estimating trace-sample miss ratios. In *Proceedings of the 19th ACM Special Interest Group on Measurement and Modeling of Computer Systems (SIGMETRICS),* Apr. 1991, 79–89.

Jason Zebchuk, Babak Falsafi, and Andreas Moshovos. 2013. Multi-grain coherence directories. In *Proceedings of the 46th IEEE/ACM International Symposium on Microarchitecture (MICRO),* Dec. 2013, 359–370.

Hongzhou Zhao, Arrvindh Shriraman, Snehasish Kumar, and Sandhya Dwarkadas. 2013. Protozoa: adaptive granularity cache coherence. In *Proceedings of the 40st International Symposium on Computer Architecture (ISCA),* June 2013, 547–558.