Noname manuscript No.
(will be inserted by the editor)

To Be Silent or Not: On the Impact of Evictions of Clean Data in Cache-Coherent Multicores

Ricardo Fernández-Pascual ·
Alberto Ros · Manuel E. Acacio

Received: date / Accepted: date

Abstract Maintaining coherence across hundreds or even thousands of cores is not an easy task. Among all of the proposed solutions until now, directory-based cache coherence has been advocated as the most feasible way of beating the scalability hurdles that arise at such large scale. Thanks to the knowledge accumulated during the last four decades, there is general consensus on the impact of most of the design aspects of directory coherence on performance, energy consumption and cost. However, there is one subtle design point for which we have observed some divergences in contemporary research works on cache-coherent multicores. Specifically, while some recent works assume a *silent* replacement policy for evictions of clean data in the last-level private caches, others implement just the opposite, that we call a *noisy* replacement policy, and even others do not mention how these evictions are managed. In this work we put this important aspect into the spotlight, demonstrating that the way in which evictions of clean data are managed can have important influence on the performance and energy consumption of a directory-based cache coherence protocol. We show that the noisy replacement policy leads to a significant increase of the total traffic (around 20% in several cases, 9.6% on average) compared with the silent policy. Given the important fraction of the total power budget that the on-chip interconnection network of future manycores is expected to consume, assuming the silent replacement policy for clean data will lead to non-negligible energy savings. Moreover, and what is more important, we have observed that depending on the particular directory structure used, assuming silent replacements could affect performance or not. This means that the use of noisy replacements is not justified in all cases, since it would increase unnecessarily network traffic without leading to any performance advantages.

Dept. de Ingeniería y Tecnología de Computadores, Universidad de Murcia, Spain
E-mail: {rfernandez,aros,meacacio}@ditec.um.es

Keywords Manycores · Cache coherence · Eviction of clean data · Directory structure · Performance · Traffic

1 Introduction and motivation

Today's mainstream multicore processors offer the shared-memory abstraction as the low-level programming paradigm. Programming systems with shared memory has been popular due to its similarity to single-threaded programming, easy mapping of certain classes of algorithms and the widespread availability of applications based on POSIX Threads and OpenMP. Additionally, operating systems has been traditionally built on the shared-memory abstraction. Unless a radical change happens (something really improbable as of today), this trend will continue at least in the immediate future [1]. Communications in these multicore architectures occur by writing to and reading from the shared memory. In order to reduce average latency of memory accesses in these architectures, as well as pressure on shared resources (interconnection network and shared cache levels), every processor core in these designs is equipped with one or more levels of private caches. A cache coherence protocol implemented in hardware is responsible for ensuring that every processor core gets always the most recent version of every memory block, avoiding incoherences and thus making caches functionally invisible to software [2].

With the rapid increase in the number of cores that are integrated on chip, the design of an efficient and scalable coherence protocol is not an easy task. Maintaining coherence across hundreds or even thousands of cores gives birth to some critical concerns that do not appear when the number of cores is low and that affects directly the cost of the final design. Also, communications in such large-scale multicores occur under the control of the cache coherence protocol, which gives it a leading role in determining both overall performance and energy requirements. Due to the increasing importance of cache coherence in the multicore era, many proposals have appeared recently in the literature trying to defeat the scalability hurdles that traditional approaches have. Most solutions towards scalable cache coherence involve the use of a distributed directory structure that holds information about which private caches maintain copies of which memory blocks. Misses at the last-level private caches are therefore sent to the corresponding directory module, which will perform the required coherence actions (e.g., invalidation of sharers on a write miss).

Since first proposed in the late 1970s [3], directory-based cache coherence solutions have been popular both in academia and in the commercial arena [2, 4]. This way, over the years, profound knowledge has been accumulated about the implications on performance, energy consumption and cost of many design decisions regarding this kind of protocols. However, although nowadays there is general consensus about the impact of most of the design aspects of directory coherence, there is one subtle design point for which we have observed some divergences in contemporary literature. It has to do with how evictions[1] of

[1] Along this work, we use the terms "replacement" and "eviction" interchangeably.

clean data in the last-level private caches are managed. Particularly, we have found[2] that some recent works assume in these situations a *silent* replacement policy [5–8], i.e., the corresponding directory module is not notified, while others implement just the opposite [9–13], that we call a *noisy* replacement policy, and even others do not mention how these evictions are managed [14, 15].

In this work we put this important aspect into the spotlight, demonstrating that the way in which evictions of clean data are managed can have important influence on the performance and energy consumption of a directory cache coherence protocol. Specifically, we show that the silent replacement policy can save a significant percentage of the total traffic budget (more than 25% in several cases, 9.6% on average) compared with the noisy one. Since the on-chip interconnection network of some contemporary multicore architectures has been shown to consume an important fraction of the total power budget (approaching 40% [16]) and future on-chip networks in many-core processors are estimated to consume hundreds of watts of power [17], assuming thus the silent replacement policy for clean data will lead to non-negligible energy savings. Moreover, and what is more important, we have observed that depending on the particular directory structure used, assuming silent replacements could affect performance or not. This means that the use of noisy replacements is not justified in all cases, since it would increase unnecessarily network traffic without leading to any performance advantages. To elaborate on this, we show different use cases in which it is better to apply each policy.

The rest of the manuscript is organized as follows. We start by discussing in Section 2 the pros and cons of assuming silent and noisy replacement policies for clean data. Subsequently, in Section 3 we present the simulation environment and the use cases that are analyzed. Then, in Section 4 we report detailed results in terms of execution time and network traffic, demonstrating the importance of correctly managing clean data evictions. Finally, Section 5 contains the main conclusions of this work.

2 Evictions of clean data: silent versus noisy

Every time a cache miss happens, the requested block is brought to cache. But, if there is no room for the block in the cache, an existing block has to be evicted to make room[3]. The replaced block may have been modified locally (a *dirty* block) or not (a *clean* block). On a replacement of a dirty block, the next cache level must be updated with the new data. Differently, on a replacement of a clean block, data at the next cache level does not need to

[2] We have performed a revision of most papers on cache coherence appeared in the last five editions of the proceedings of ISCA, HPCA, PACT and MICRO conferences, and we have found that, out of 36 papers, noisy replacements are assumed in 14, silent replacements in 8, and 14 papers do not mention the used policy.

[3] Consequently, replacements are almost as frequent as cache misses once the cache hierarchy is warmed up.

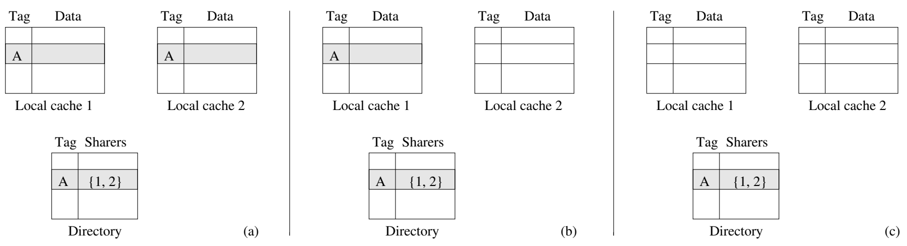


Fig. 1: Example of silent evictions of clean data.

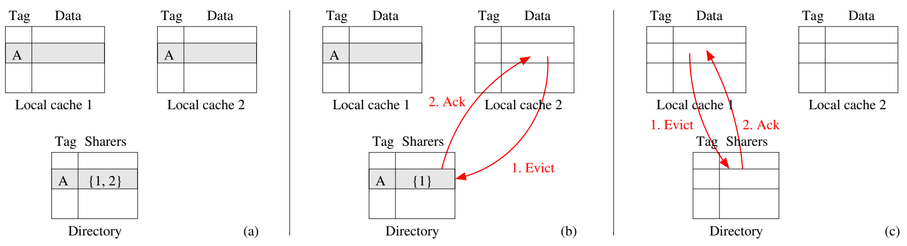


Fig. 2: Example of noisy evictions of clean data.

be updated although, as we explain in this section, it may be desirable to inform the directory about clean evictions at the private cache level so that the sharing information of the block can be updated.

Thus, an important decision when designing a directory-based cache coherence protocol is how to handle evictions of clean data. One option is to replace the data block without informing the directory (known as a *silent* eviction). The other option is to inform the directory about the replacement (what we call a *noisy* eviction).

2.1 Examples of silent and noisy evictions

Two examples of the behavior of silent and noisy evictions are shown in Figures 1 and 2, respectively. Without loss of generality, the examples consider two local caches and a single directory bank tracking the blocks stored in the caches. In both cases, the initial situation *(a)* is the same: there are two caches holding clean copies of the block A and the directory tracks their identity in a *sharers* field. This field could be implemented using a bit-vector or any other exact sharing code.

In Figure 1, evictions are performed in a silent way. Column *(b)* shows the eviction of block *A* from *local cache 2*. For this, no message needs to be exchanged between any nodes. The local cache simply discards the data. The directory, therefore, keeps considering that *local cache 2* still has the data.

It is important to remember that no incorrect behaviour will happen if the directory does not have exact information about the set of sharers as long as the sharing code includes all current sharers. That is, it can include also nodes that are not currently sharing the block). If the directory needs to invalidate the sharers of the block, those nodes will receive an (unnecessary) invalidation message which will need to be answered but it does not affect correctness (it leads to extra traffic and latency as we will explain in next subsection).

Column (c) of the same Figure 1 shows another silent eviction for the same block, this time from *local cache 1*. In the end, the entry for block A is kept in the directory even though no local cache is using that block anymore.

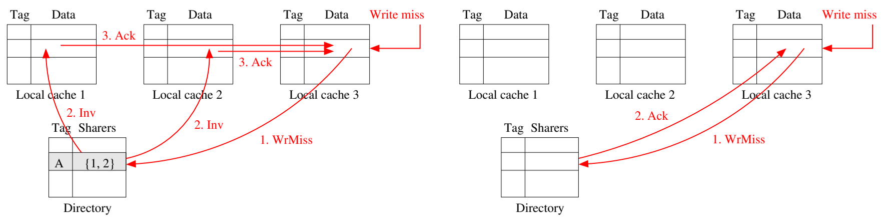
On the other hand, Figure 2 illustrates how to perform the two same evictions in a noisy way. In this case, messages need to be exchanged between the directory and the replacing cache. First, an eviction notification is sent to the directory. On receiving this, the directory updates the sharing information of the block removing *local cache 2* from the set of sharers. Finally, the directory acknowledges *local cache 2* about the replacement and the transaction finishes.

Again, column (c) of Figure 2 shows a second replacement for the same block which is carried out exactly as the previous one. But, unlike in Figure 1, since noisy replacements are being used this time, the directory updates the sharing information of the block and when it notices that there are no more cached copies of block A, it frees the directory entry occupied by that block.

2.2 Consequences of clean data eviction policy

The first consequence of the silent-versus-noisy eviction decision is the amount of network *traffic* generated by the cache coherence protocol. Informing the directory about the eviction of a clean block entails two control messages, as shown in Figure 2. Observe that although two control messages would also be required for invalidating a replaced clean copy if evictions are instead silent, this would only be paid for written blocks and blocks whose directory entry is evicted from the directory structure. Therefore, read-only blocks or clean blocks that are replaced often will benefit from a silent eviction policy. Since we have observed that these situations are frequent, the extra traffic generated by noisy evictions can represent a large fraction of the overall coherence traffic.

The second consequence of implementing noisy evictions for clean data is the *accuracy* of the directory information. This affects the coherence protocol in two different ways. On one hand, the number of copies tracked by the directory is reduced. This means that upon a write miss or a directory replacement fewer invalidation messages are issued, and consequently fewer acknowledgments need to be received. Issuing fewer invalidations reduces both traffic requirements and write miss latency. Reporting replacements of clean data could be also beneficial when each directory entry can only track a limited number of sharers (i.e. limited pointers sharing codes), since it would naturally recycle directory information, saving overflow situations that otherwise may arise. On the other hand, and more importantly, the directory occupation is reduced because directory entries for blocks that have been evicted from all the local caches can be deallocated. Reducing the pressure in the directory



(a) Write miss when using silent evictions. (b) Write miss when using noisy evictions.

Fig. 3: Example of a write miss under different eviction policies.

leads to fewer directory replacements and, consequently, fewer invalidations which leads to fewer cache misses.

The effect of having accurate directory information when handling write misses is depicted in Figure 3. Assume for example that a write miss happens for block A after the two replacements shown in Figures 1 and 2. In the case of silent replacements (Figure 3a), the directory has to send invalidation messages to the L1 caches even though they do not hold any copy of the block at that moment because the directory was not notified when the block was evicted, thus generating traffic and incurring in extra latency in the critical path of the write miss. However, in the case of noisy replacements (Figure 3b), the directory does not need to perform unnecessary invalidations and it can directly give permission to write to the requesting cache.

There is a third consequence of the eviction policy mentioned by Sorin *et al.* [2] related to the implementation complexity of the protocol. Silent evictions allow a race condition when a block is requested just after having been evicted from the same cache. If soon after that the cache receives an invalidation due to a write request of another node, there is no straightforward way to know if the invalidation should take place before or after receiving the requested data. This complexity is the reason why Sorin *et al.* opt for noisy evictions. However, although complex solutions to determine the most appropiate way to solve this race may increase the protocol complexity, there is a very simple option which is to always invalidate the data when it arrives. Since this race happens rarely, more efficient solutions are not necessary. Thus, in practice both evictions policies require similar complexity, and as a consequence of this, we believe that the appropriate choice should be driven by performance.

2.3 When to use silent or noisy evictions

From the previous examples we can deduce that both kinds of eviction policies have advantages and drawbacks. Silent evictions can reduce coherence traffic due to replacements, while noisy evictions can reduce write miss latency and improve directory efficiency by requiring fewer directory entries. The best policy depends both on the application characteristics and on the cache coherence protocol employed.

The drawbacks of silent evictions are the extra invalidations when handling write misses and the extra directory evictions caused by the reduced precision of the sharing information. Read-only data, therefore, will benefit from silent evictions if they are frequently evicted and re-fetched before the directory needs to replace the corresponding entry. Similarly, read-write data would be favored by the silent evictions if the number of read misses per cache is much larger than the number of write misses. Otherwise, noisy evictions may be preferable due to the lower latency of write misses.

The properties of the cache coherence protocol employed have even more importance to take a decision about the use of silent or noisy replacements. For example, a linked-list protocol [18–20] would not be able to employ silent replacements in any case because the list of sharers needs to be updated [21]. Similarly, a token-based protocol [22, 23] cannot use silent invalidations either because the tokens held by a sharer need to be transferred to the directory.

The decision becomes specially relevant for protocols that employ directory caches where the number of entries used per block depends on the number of sharers (e.g., dynamic pointer allocation [24] or SCD [13]). In these cases, the better precision provided by noisy replacements can allow for better directory usage and, consequently, reduce the number of directory replacements and the amount of information each directory entry must track. This is less important in more traditional directories because, in that case, directory entries can be deallocated only when the number of sharers reaches zero.

3 Evaluation Methodology

We evaluate the use of silent and noisy replacements for different cache coherence protocols using the PIN [25] and GEMS 2.1 [26] tools, which have been connected in a similar way as described by Monchiero *et al.* [27]. In particular, PIN obtains the instructions executed and the memory references performed by them along with all the synchronization primitives employed in the applications, while in GEMS we model an in-order core that issues memory requests to the GEMS' memory hierarchy to calculate the access latency for each processor request. We model the interconnection network with the SiCoSys [28] simulator.

The simulated architecture corresponds to a single chip multiprocessor (*tiled*-CMP) with 64 cores. The directory and the shared L2 cache banks are distributed following a per-block interleaving across the 64 tiles. The memory controllers are placed on chip (one per tile) and access and off-chip banked memory. The remaining simulation parameters are shown in Table 1.

We evaluate the impact that the implementation of evictions of clean data has on two configurations of a 64-core CMP architecture. The first one employs a sparse directory using non-scalable bit-vectors in each directory entry as the sharing code (first bar in all graphs of Section 4 assumes silent evictions and

Table 1: System parameters.

Memory parameters	
Block size	64 bytes
L1 cache (data & instr.)	32 KiB, 4 ways
L1 access latency	1 cycle
L2 cache (shared)	256 KiB/tile, 16 ways
L2 access latency	6 cycle
Cache organization	Inclusive
Directory information	Sparse with 512 entries, 4 ways (100% coverage)
Memory access time	160 cycles
Network parameters	
Topology	2-D mesh (8×8)
Switching and Routing	Wormhole and X-Y
Message size	4 flits (data), 1 flit (control)
Link time	2 cycles
Bandwidth	1 flit per cycle

second bar noisy ones). The second configuration implements the recently proposed SCD directory architecture [13] (third bar assumes silent evictions and fourth bar noisy ones).

Both coherence protocols implement local caches with MESI states. Our evaluation considers only clean-versus-noisy evictions of blocks in S (shared) state. We assume that the E (exclusive) state is implemented as an ownership state, and therefore, silent evictions are not possible [2]. Additionally, the number of replacements in E state compared to S state is negligible, as it is shown in the evaluation. Alternatively, the E state can be implemented as a non ownership state (e.g., like OpenPiton [29]) which would allow using silent replacements too.

Our simulations consider representative applications from both the Splash-2 [30] and the PARSEC 2.1 [31] benchmark suites. *Barnes*, *Cholesky*, *FFT*, *Ocean*, *Radix*, *Raytrace*, *Volrend*, and *Water-NSQ* use the input sizes used in the Splash-2 paper. *Bodytrack*, *Canneal*, *Streamcluster*, and *Swaptions* are from the PARSEC 2.1 suite and use the *simmedium* input sizes. We have accounted for the variability of parallel applications as discussed in [32]. To do so, we have performed a number of simulations for each application and configuration inserting random variations in each main memory access. All results in this work correspond to the parallel part of the applications.

4 Results

The aim of our evaluation is to show the impact of silent versus noisy evictions. The evaluation is performed on two directory organizations (*BitVector* and *SCD*) where the election of the eviction policy can affect the performance in different ways.

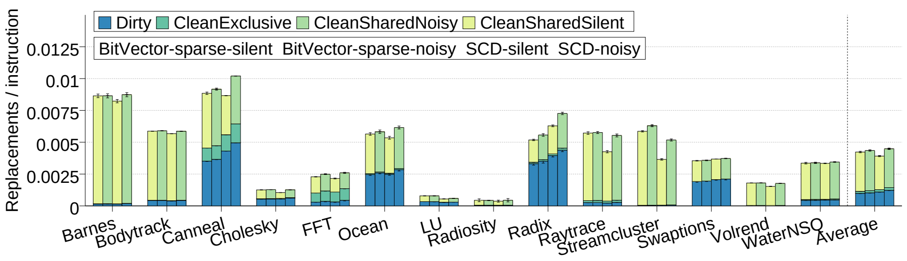


Fig. 4: L1 replacements per instruction, categorized.

Frequency and characterization of L1 replacements. Our first analysis focuses on the frequency of L1 cache replacements considering each type of evicted block. For each application, Figure 4 plots the number of L1 cache replacements per instruction executed. Replacements are classified in four categories: *Dirty*, *CleanExclusive*, *CleanSharedNoisy*, and *CleanSharedSilent*. Dirty replacements correspond to blocks in M (modified) state. CleanExclusive replacements correspond to blocks in E state. Replacements of blocks in S state are labeled as CleanSharedNoisy when the protocol implements noisy evictions or as CleanSharedSilent when the protocol implements silent evictions.

On average, blocks in E state are not involved in a significant fraction of the replacements. Exceptions are applications like *Canneal* and *FFT*. However, most of the L1 cache replacements (67% for SCD and 73% for a bit-vector directory, on average) correspond to blocks in S state. This is the first indicator of the importance that implementing this kind of replacements in the most efficient way may have. Also, we see an increase of L1 replacements in some cases when using noisy replacements. This is a consequence of the reduction in directory replacements when using the noisy policy that we explain in the following paragraph (i.e., with silent evictions more blocks are invalidated due to directory replacements before they need to be evicted from L1).

Frequency of directory replacements. Noisy evictions increase the accuracy of the information at the directory by removing from the set of tracked sharers those nodes that evict the block from their local cache. In a traditional sparse directory using a bit-vector sharing code, directory entries are only deallocated when the count of sharers reaches zero. However, in directories where the sharing code is distributed among different entries, like in SCD, the use of noisy replacements helps to reduce directory pressure because it allows to deallocate some entries as soon as they become useless, which will reduce the directory occupancy. With the aim of analyzing this effect, Figure 5 plots the number of directory replacements per instruction executed. On average, when using noisy replacements the number of directory replacements can be reduced by 41% for a bit-vector directory and by 66% for a SCD directory.

Impact on L1 write miss latency. Another expected effect of noisy replacements is the reduction in the latency of write misses because fewer invalidations need to be sent. Figure 6 plots the latency of write misses with each

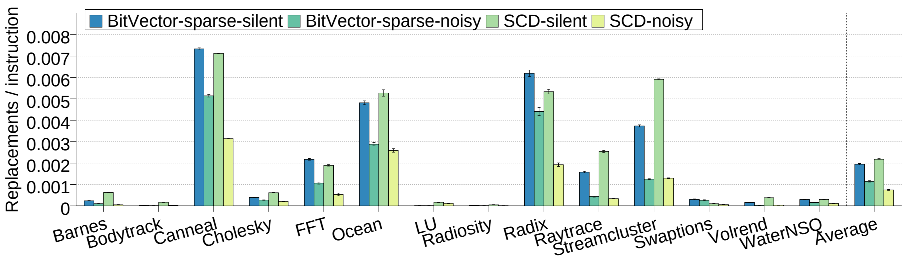


Fig. 5: Directory replacements per instruction.

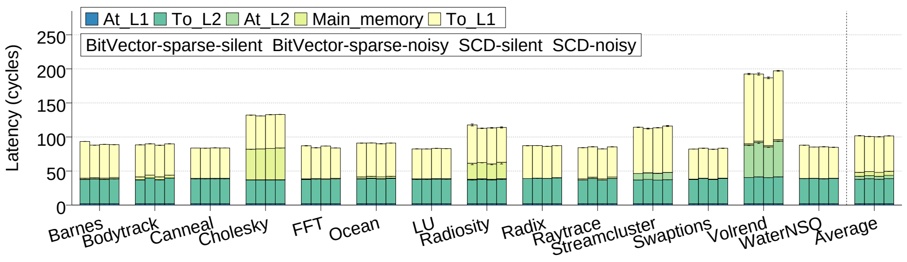


Fig. 6: L1 write miss latency.

protocol. Each bar in this figure has been split in five parts: the time spent in accessing the L1 cache (*At_L1*); the time spent while the requests travels from the L1 to the L2 (*To_L2*) required to access the directory information; the time spent waiting until the L2 can attend the miss (*At_L2*), mostly due to on-going transactions on the same memory block; the time spent waiting to receive the data from main memory (*Main_memory*) in case the requested block is not present in any on-chip cache; and the time spent since the moment that the L2 sends the data or forwards the request until the requester receives the data and every required acknowledgment and the miss is resolved (*To_L1*). In fact, we only see very small variations in latency: it decreases very slightly for some benchmarks in the case of the bit-vector directory and increases also very slightly for other benchmarks in the case of SCD. This is so because in practice the number of invalidations per miss is similar and, moreover, all invalidations are sent in parallel and only the processing of the acknowledgments actually benefits, because it needs to be done serially by the requester. On the other hand, the additional traffic and directory replacements explain the slight increase in latency.

Impact on L1 read miss latency. On the other hand, the effect on the latency of read misses can be seen in Figure 7. Read misses are handled exactly the same whether noisy or silent evictions are used, and any differences in latency should be explained primarily by the different traffic that travels through the network and by the different number directory replacements due to the increased accuracy led by the noisy replacements policy with respect to the silent replacements one. For example, the read miss latency of *Streamcluster*

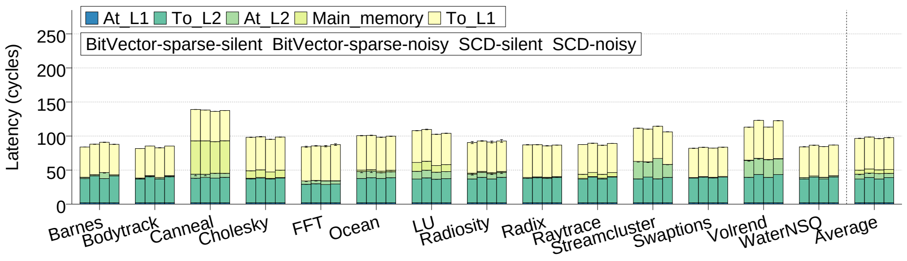


Fig. 7: L1 read miss latency.

with SCD is reduced when using noisy replacements versus silent replacements. This is because the *At_L2* time is smaller, which happens as a consequence that there are fewer directory replacements keeping the directory controller busy. As in the case of writes, on average we see that the differences are minimal, with an almost insignificant increase in the case of noisy replacements.

Impact on network traffic. The major impact of the replacement policy for clean data is on network traffic. An increase in traffic will directly translate into an increase in energy consumption in the interconnection network. Figure 8 shows the coherence traffic measured in flits and normalized with respect to a bit-vector implementing silent evictions. Traffic has been divided in the following categories: data messages due to cache misses (*Data*), data messages due to replacements (*WBData*), control messages due to cache misses (*Control*), control messages due to replacements (*WBControl*). We see that, although a protocol with silent replacements generates in some cases a bit more control messages due to cache misses, a protocol with noisy replacements generates always significantly more control messages due to replacements. In the case of the bit-vector protocol, noisy evictions almost always increase traffic overall (except in *FTT* and *Radix*), but the increase is less sharp in the case of SCD and there are more exceptions (*FTT* and *Radix* again, but also *Canneal*, *Cholesky*, *LU*, and *Streamcluster*). This is because noisy evictions can reduce the number of directory replacements (as shown in figure 5). These directory replacements require the invalidation of any copy of the block in any local cache, so cutting down on them reduces cache misses which in turn reduces the amount of control messages and, more importantly, data messages (which are longer in size). On average, the noisy evictions policy increases the total network traffic by 9.6% for a bit-vector directory and by 4.1% for a SCD directory.

Impact on accesses to directory and L1 cache. The replacement policy also affects the number of accesses that need to be performed to the directory and the L1 cache, as can be seen in Tables 2 and 3. An increase in accesses to these cache structures will increase the energy consumption in the cache hierarchy. In general, noisy replacements increase the number of accesses to the directory because the directory needs to be accessed to update the sharing code on each replacement. Silent replacements slightly increase the

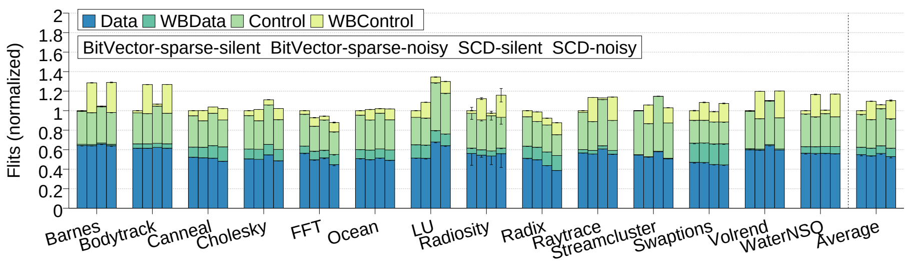


Fig. 8: Network traffic.

	Barnes		Bodytrack		Canneal		Cholesky		FFT		Ocean		LU	
	L1	**Dir**	**L1**	**Dir**	**L1**	**Dir**	**L1**	**Dir**	**L1**	**Dir**	**L1**	**Dir**	**L1**	**Dir**
BV-silent	638.15	11.45	2018.81	41.43	513.02	54.84	425.18	6.49	8.55	0.37	222.67	13.45	247.76	1.41
BV-noisy	637.31	21.02	2005.31	71.55	506.34	58.67	425.04	7.24	8.57	0.37	223.68	14.14	248.07	1.67
SCD-silent	651.46	12.13	2008.81	42.26	518.79	58.85	429.38	7.19	8.53	0.35	223.76	13.81	248.09	1.65
SCD-noisy	636.35	21.11	2000.59	71.42	510.31	60.86	423.13	7.27	8.68	0.36	220.15	14.23	247.92	1.76

	Radiosity		Radix		Raytrace		Streamcluster		Swaptions		Volrend		WaterNSQ	
	L1	**Dir**	**L1**	**Dir**	**L1**	**Dir**	**L1**	**Dir**	**L1**	**Dir**	**L1**	**Dir**	**L1**	**Dir**
BV-silent	802.49	2.93	21.66	1.45	550.09	10.77	3661.11	111.15	3085.28	85.45	1885.78	9.60	145.40	2.59
BV-noisy	803.82	4.28	21.04	1.45	555.38	15.69	3599.33	137.62	3084.93	98.86	1900.07	16.63	145.23	3.72
SCD-silent	794.79	2.93	20.36	1.37	570.57	12.49	3762.59	134.15	3084.14	85.60	1885.26	11.17	145.73	2.62
SCD-noisy	799.82	4.41	20.16	1.33	561.61	15.42	3606.13	127.52	3084.15	98.87	1902.15	16.54	145.21	3.74

Table 2: Millions of accesses per application to the directory and L1 cache structures.

	L1 Cache	**Directory**
BV-silent	1016.14	25.24
BV-noisy	1011.72	32.35
SCD-silent	1025.16	27.61
SCD-noisy	1011.88	31.77

Table 3: Millions of accesses (average) to the directory and L1 cache structures.

number of accesses to the L1 cache due to extra invalidation messages received upon write or directory eviction. The total number of accesses, dominated by L1 caches, does not vary significantly for any directory scheme and replacement policy (cache accesses are traded by almost the same number of directory accesses). Considering that the accesses to the L2 caches do not vary, we can conclude that the energy consumption in the memory hierarchy is not affected to a great extent by the eviction policy (we assume that directory accesses have a similar cost to L1 accesses, which is true if both have the same associativity and number of entries).

Impact on execution time. We also show the impact on execution time of the replacement policy in Figure 9. All results have been normalized again to a bit-vector implementing silent evictions. On average, the use of noisy replacements does not affect at all the performance of a bit-vector directory when compared to silent replacements. Since noisy replacements increase traffic requirements by 9.6%, we can conclude that coherence protocols using bit-vector

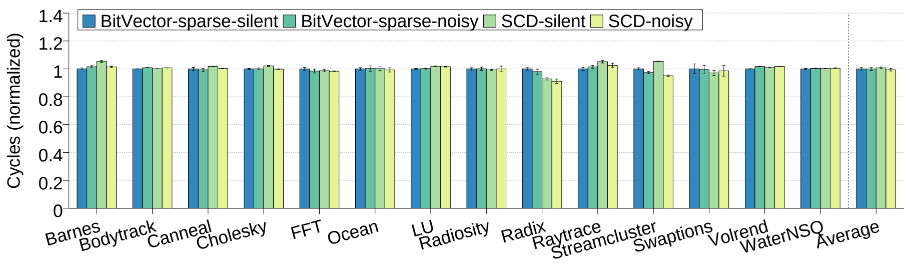


Fig. 9: Execution time.

directories are better implemented with silent replacements. The effect in the average execution time of SCD is very small: noisy replacements reduce the execution time by 1.5% at the cost of a 4.1% increase in traffic. Therefore, using noisy replacements in a protocol like SCD may be justified if the designer wants to achieve the maximum performance regardless of energy efficiency.

Scalability. To conclude the evaluation, we have also performed simulations of the replacement policies for smaller systems (i.e., 16 cores), although we do not plot the results for the sake of brevity. For a 16-core system using a bit-vector directory, noisy replacements do not affect the execution time while they increase traffic requirements by 8% on average and up to 24% in the worst case (Water-NSQ). In the case of SCD, noisy replacements obtain also the same execution time as silent replacements, while traffic increases by 1.6% on average and up to 17% in the worst case (Water-NSQ). This shows that noisy replacements in SCD become more useful for higher core counts, and that the differences in execution time between the two policies are also more important when the system scales to higher core counts.

5 Conclusions

In this work we focus on a frequently downplayed design aspect of a directory-based cache coherence protocol: the way that evictions of clean data are managed. We discuss the two alternatives that are possible (managing them silently versus noisily), highlighting the pros and cons of each one. Then, we consider two scenarios in which each policy would be preferable, and compare them taking into account performance (execution time) and amount of network traffic.

We found that, depending on the characteristics of the directory structure, this aspect may have negligible effect on performance or can impact execution time. The latter happens when the resources of the directory limit the number of shares per entry or the total number of addresses that can be present in the directory. In all cases, the noisy replacement policy increases network traffic when compared to the silent one.

The main conclusion of this study is that a cache coherence protocol implementing bit-vector directories should also implement silent evictions, which is not always the preferred option in the literature, while cache coherence protocols implementing a directory with a sharing code that uses a variable number of directory entries can benefit from noisy evictions.

Specifically, we find out that assuming the noisy replacements policy for a bit-vector directory is not justified at all. It would increase network traffic more than 25% in several cases (9.6% on average) without benefiting execution time in any way. The fact that all sharers must have replaced their copies of the block before the directory entry can be deallocated leaves very few opportunities in which a directory entry can actually be reused thanks to the eviction notifications. Conversely, the fact that many clean blocks are replaced several times by each processor before invalidation (due to write misses or evictions of directory entries) creates so much extra traffic that at the end noisy evictions would result in increased energy consumption. This way, one important conclusion we would like to draw is that studies that use a bit-vector directory as the baseline to compare with should assume the silent replacements policy. Otherwise, they would be comparing against a sub-optimal baseline.

On the other hand, we see that the ability of noisy evictions of increasing the accuracy of the information at the directory structure has balsamic effects in some applications for directories where the sharing code is distributed among different entries, like in SCD. In these cases, the use of noisy replacements helps to reduce directory pressure since it allows to deallocate some entries as soon as they become useless (the few sharers they were tracking have replaced their copy of the block). This translates into noticeable reductions in directory occupancy. Thanks to this effect, we have seen that execution time in SCD can be reduced up to 9.8% (2.7% on average) when the noisy evictions policy is applied.

To summarize, the aim of this manuscript is to demonstrate that the decision about using silent or noisy evictions depends on the kind of directory structure being used, and that its effects are noticeable, so that designers should consider it as a first-class design decision.

Acknowledgments

This work has been supported by the Spanish MINECO, as well as European Commission FEDER funds, under grant "TIN2015-66972-C5-3-R" and by the Fundación Séneca-Agencia de Ciencia y Tecnología de la Región de Murcia under grant "19295/PI/14".

References

1. M. M. K. Martin, M. D. Hill, D. J. Sorin, Why on-chip cache coherence is here to stay, Communications of the ACM 55 (7) (2012) 78–89.
2. D. J. Sorin, M. D. Hill, D. A. Wood, A Primer on Memory Consistency and Cache Coherence, Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2011.

3. L. M. Censier, P. Feautrier, A new solution to coherence problems in multicache systems, IEEE Transactions on Computers (TC) 27 (12) (1978) 1112–1118.
4. D. E. Culler, J. P. Singh, A. Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann Publishers, Inc., 1999.
5. D. Vantrease, M. H. Lipasti, N. Binkert, Atomic coherence: Leveraging nanophotonics to build race-free cache coherence protocols, in: 17th Int'l Symp. on High-Performance Computer Architecture (HPCA), 2011, pp. 132–143.
6. B. Cuesta, A. Ros, M. E. Gómez, A. Robles, J. Duato, Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks, in: 38th Int'l Symp. on Computer Architecture (ISCA), 2011, pp. 93–103.
7. M. Elver, V. Nagarajan, TSO-CC: Consistency directed cache coherence for tso, in: 20th Int'l Symp. on High-Performance Computer Architecture (HPCA), 2014, pp. 165–176.
8. M. Zhang, J. D. Bingham, J. Erickson, D. J. Sorin, PVCoherence: Designing flat coherence protocols for scalable verification, in: 20th Int'l Symp. on High-Performance Computer Architecture (HPCA), 2014, pp. 392–403.
9. J. Zebchuk, B. Falsafi, A. Moshovos, Multi-grain coherence directories, in: 46th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO), 2013, pp. 359–370.
10. S. Demetriades, S. Cho, Stash directory: A scalable directory for many-core coherence, in: 20th Int'l Symp. on High-Performance Computer Architecture (HPCA), 2014, pp. 177–188.
11. L. G. Menezo, V. Puente, J.-Á. Gregorio, Flask coherence: A morphable hybrid coherence protocol to balance energy, performance and scalability, in: 21th Int'l Symp. on High-Performance Computer Architecture (HPCA), 2015, pp. 198–209.
12. M. Zhao, D. Yeung, Studying the impact of multicore processor scaling on directory techniques via reuse distance analysis, in: 21th Int'l Symp. on High-Performance Computer Architecture (HPCA), 2015, pp. 590–602.
13. D. Sanchez, C. Kozyrakis, SCD: A scalable coherence directory with flexible sharer set encoding, in: 18th Int'l Symp. on High-Performance Computer Architecture (HPCA), 2012, pp. 129–140.
14. G. Zhang, W. Horn, D. Sanchez, Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems, in: 48th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO), 2015, pp. 13–25.
15. Y. Fu, T. M. Nguyen, D. Wentzlaff, Coherence domain restriction on large scale systems, in: 48th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO), 2015, pp. 686–698.
16. T. Moscibroda, O. Mutlu, A case for bufferless routing in on-chip networks, in: 36th Int'l Symp. on Computer Architecture (ISCA), 2009, pp. 196–207.
17. S. Borkar, Thousand core chips: a technology perspective, in: 44th Design Automation Conference (DAC), 2007, pp. 746–749.
18. D. V. James, A. T. Laundrie, S. Gjessing, G. S. Sohi, Scalable coherent interface, IEEE Computer 23 (6) (1990) 74–77.
19. T. Lovett, R. Clapp, STiNG: A cc-NUMA computer system for the commercial marketplace, in: 23rd Int'l Symp. on Computer Architecture (ISCA), 1996, pp. 308–317.
20. R. Thekkath, A. P. Singh, J. P. Singh, S. John, J. L. Hennessy, An evaluation of a commercial cc-NUMA architecture: The CONVEX Exemplar SPP1200, in: 11th Int'l Symp. on Parallel Processing (IPPS), 1997, pp. 8–17.
21. R. Fernández-Pascual, A. Ros, M. E. Acacio, Optimization of a linked cache coherence protocol for scalable manycore coherence, in: 29th Int'l Conf. on Architecture of Computing Systems (ARCS), 2016, pp. 100–112.
22. M. M. Martin, M. D. Hill, D. A. Wood, Token coherence: Decoupling performance and correctness, in: 30th Int'l Symp. on Computer Architecture (ISCA), 2003, pp. 182–193.
23. M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. Martin, D. A. Wood, Improving multiple-CMP systems using token coherence, in: 11th Int'l Symp. on High-Performance Computer Architecture (HPCA), 2005, pp. 328–339.
24. R. Simoni, M. A. Horowitz, Dynamic pointer allocation for scalable cache coherence directories, in: Int'l Symp. on Shared Memory Multiprocessing, 1991, pp. 72–81.
25. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, Pin: Building customized program analysis tools with dynamic instrumentation, in: 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), 2005, pp. 190–200.

26. M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, D. A. Wood, Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset, Computer Architecture News 33 (4) (2005) 92–99.
27. M. Monchiero, J. H. Ahn, A. Falcón, D. Ortega, P. Faraboschi, How to simulate 1000 cores, Computer Architecture News 37 (2) (2009) 10–19.
28. V. Puente, J. A. Gregorio, R. Beivide, SICOSYS: An integrated framework for studying interconnection network in multiprocessor systems, in: 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing, 2002, pp. 15–22.
29. J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, D. Wentzlaff, Openpiton: An open source manycore research framework, in: 21st Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS), 2016, pp. 217–232.
30. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta, The SPLASH-2 programs: Characterization and methodological considerations, in: 22nd Int'l Symp. on Computer Architecture (ISCA), 1995, pp. 24–36.
31. C. Bienia, S. Kumar, J. P. Singh, K. Li, The PARSEC benchmark suite: Characterization and architectural implications, in: 17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT), 2008, pp. 72–81.
32. A. R. Alameldeen, D. A. Wood, Variability in architectural simulations of multi-threaded workloads, in: 9th Int'l Symp. on High-Performance Computer Architecture (HPCA), 2003, pp. 7–18.