Flask Coherence: A Morphable Hybrid Coherence Protocol to Balance Energy, Performance and Scalability

Lucia G. Menezo Valentin Puente Jose-Angel Gregorio
University of Cantabria
Santander, Spain
{gregoriol, vpuente, monaster}@unican.es

Abstract—**This work proposes a mechanism to hybridize the benefits of snoop-based and directory-based coherence protocols in a single construct. A non-inclusive sparse-directory is used to minimize energy requirements and guarantee scalability. Directory entries will be used only by the most actively shared blocks. To preserve system correctness token counting is used. Additionally, each directory entry is augmented with a counting bloom filter that suppresses most unnecessary on-chip and off-chip requests. Combining all these elements, the proposal, with a low storage overhead, is able to suppress most traffic inherent to snoop-based protocols. With a directory capable of tracking just 40% of the blocks kept in private caches, this coherence protocol is able to match the performance and energy of a sparse-directory capable of tracking 160% of the blocks. Using the same configuration, it can outperform the performance and on-chip memory hierarchy energy of a broadcast-based coherence protocol such as Token by 10% and 20% respectively.**

To achieve these results, the proposal uses an improved counting bloom filter, which provides twice the space efficiency of a conventional one with similar implementation cost. This filter also enables the coherence controller storage used to track shared blocks and filter private block misses to change dynamically according to the data-sharing properties of the application. With only 5% of tracked private cache entries, the average performance degradation of this construct is less than 8% compared to a 160% over-provisioned sparse-directory.

Keywords – coherence protocol; multi-core; CMPs

I. INTRODUCTION

The enforcement of hardware coherence in contemporary CMPs with complex on-chip cache hierarchy constitutes an interesting problem of competing trade-offs in cost, energy and performance. The solutions adopted for this problem vary greatly from system to system, it being unclear whether there is a universal solution. To achieve the expected scalability of future many-core CMPs, the use of a directory-based coherence protocol seems unavoidable. In contrast, current high-performance commercial systems seem to favor the use of broadcast-based coherence protocols [9], [14], [16]. This approach is used sometimes for the intra-chip and off-chip realm even with a large number of coherent cores. Consequently, although from the energy standpoint, broadcast-based coherence loses its appeal when the number of cores in the system grows, its performance advantage and complexity compared to directory-based coherence makes it predominant.

This work was supported by the Spanish Government, under contract TIN2013-43228-P, the University of Cantabria, under grant VP07, and by the HiPEAC European Network of Excellence.

In general, directory-based approaches demand inclusivity to guarantee correctness. When the number of cores grows, simple approaches such as duplicate-tag directories become inefficient due to the large associativity required. To meet energy constraints, the solution adopted is to over-provision the directory to minimize the unnecessary evictions in private caches due to directory conflicts under a constrained (and realistic) associativity [15]. Unfortunately, the size of the private caches is growing as a consequence of the larger and larger sizes in the last-level cache (LLC), which implies that the number of blocks the directory must track has to be larger.

On the other hand, broadcast-based coherence protocols interrogate all coherence agents in the chip when a core misses the desired block in its private caches. In order to guarantee correctness, neither inclusivity nor additional structures to track block copies are required. Therefore, resource utilization is better. Nevertheless, this is achieved at the cost of increasing the traffic and cache snoops, which will decrease the energy efficiency of the system. In small-scale systems, this might be tolerable, but when the size of the system grows, the impact will be noticeable, and possibly unsustainable. A more subtle effect, but a no less relevant one, is the on-chip resource contention that characterizes these protocols. As a result, on-chip access latency can be affected, perhaps degrading the CMP performance under some particular usage scenarios.

From this standpoint, it would appear that a pure coherence protocol might not be the most suitable approach to tackle the problem. Intuitively, it seems that the coherence protocol should somehow hybridize the best of both types: trying to attain the performance effectiveness and implementation cost of a broadcast-based coherence protocol with the energy efficiency of a directory-based one. This paper addresses this task and successfully attains a new coherence protocol, denoted FLASK (FiLtered and Allocated just by Shared block Keeper) coherence, which can scale as a directory-based coherence protocol does, while achieving cache effectiveness similar to a broadcast-based one.

FLASK combines three components in a single logic substrate. It uses a directory to track blocks that are *actively shared*. Therefore, private blocks, which are the most frequent case, never allocate an entry in the directory unless they are accessed during their cache lifetime (i.e., from miss to eviction) by another core. Moreover, the inclusiveness property (both for

directory and LLC) is not required, avoiding external private cache invalidations due to directory conflicts. Correctness is guaranteed through token counting. When the copies of a shared block are not being tracked by the directory, after a new request to the coherence controller a broadcast to all coherence agents is generated. The replies are used to reconstruct the directory entry. This approach to perform reconstruction of directory entries on demand was introduced by MOSAIC [27]. Nevertheless, MOSAIC allocates directory entries for any on-chip miss (i.e., for both private and shared blocks) and always generates a broadcast if there is a miss in the directory. To minimize unnecessary snoops, each entry in the directory is assisted by a filter that suppresses most unnecessary snoops. This component can directly identify the majority of on-chip misses. As a consequence, nearly all of the requests to off-chip coherence agents (memory controllers and/or off-chip coherence fabric) are not delayed. In the least common case, misses in a private cache of an actively shared block are always tracked by the filter and dealt with through a multicast to the on-chip coherence agents (private cache coherence controllers). The filter also has to avoid unnecessary off-chip requests. Finally, the architecture of the filter proposed allows us to dynamically assign, according to the sharing degree of the running workload, storage capacity in the coherence controller either to track shared blocks in the directory or to identify privately held blocks. This construct allows us to reduce the size of the directory even further.

The main contributions of the paper are:

- The hybridization of directory and a broadcast coherence in a unified logic substrate with optimized implementation and energy costs.
- The proposed strategy can achieve the performance of a conventional over-provisioned sparse directory, while tracking less than 40% of the private cache entries. Similarly, it improves on Token coherence protocol performance by 10% and energy delay product by 20%.
- With only 5% of tracked private cache entries, average performance degradation is less than 8% with respect to a 160% over-provisioned sparse-directory.
- We show that, using an adaptive storage assignation at the coherence controller according to the workload properties, we can reduce even further the resources of the directory. This is based on the distinctive properties of the filter mechanism used. Under these circumstances, the proposal is able to match a sparse-directory performance while tracking only 20% of private cache entries.

II. BACKGROUND AND MOTIVATION

A. Directory Coherence Shortcomings

Directory-based protocols seem to be an attractive approach to enforce cache coherence in a CMP. Nevertheless, when the number of cores is high and the on-chip hierarchy complexity grows, the directory is difficult to scale. We will assume a multilevel hierarchy with a shared last-level cache (LLC). A cost effective way to track the coherence information in this structure is to use a sparse-directory [15]. In contrast to in-cache directory, only the data actively used by the cores (i.e. contained in private caches) has to be tracked. This is much more cost effective, since the storage devoted to LLC is usually larger than private caches. Additionally, LLC has to be banked in order to alleviate access contention. In most cases, it is appealing to use scalable interconnects to connect these banks [17]. Under these circumstances, it is straightforward to bank the directory accordingly and attain an easy-to-handle, distributed structure.

In an on-chip cache, similar to state-of-the-art systems [10], [14], [16], in order to close the gap in the access time between a small L1 (dominated by processor clock cycle) and a very large LLC (dominated by main memory access time), an intermediate level is required. As a consequence the number of blocks that the directory has to track is larger. Additionally, those intermediate levels usually have a substantial associativity. Recent designs [14][16] also require a large associativity for L1. In summary, the number of blocks that can be mapped in a set of the directory could be high. Although in some early CMPs [21] the directory is provisioned to keep all the blocks in the private caches, when the number of cores or private cache complexity and size grows, this is not feasible due to the enormous associativity required by the directory. However, reducing this associativity increases the eviction of blocks in the private caches due to conflicts in the directory.

A rule of thumb [15] suggests that over-provisioning the directory with twice the capacity required to track the private caches will diminish the problem. Since the amount of cache to track and the sharing vector will increase with the number of cores, the cost of the directory grows quadratically. Otherwise, the number of invalidations in the private caches would grow significantly [12], impairing system performance.

Fortunately, the directory cost problem can be tackled by considering the application semantics. It is known that most memory regions are accessed privately by a single core most of the time [11][3]. If the directory is aware (actively [11][3] or passively [27][12]), we can reduce the number of private blocks that we have to track in the directory. Consequently, we can reduce the number of entries without interfering with the private cache performance.

B. Broadcast Coherence Shortcomings

The main problem with these protocols is their scalability issues, due to the extra traffic and cache snoops that each private cache miss triggers. Similarly to the directory, this might grow when the number of cores is increased (because more traffic is required) and the private caches are bigger (because the tag snoops are more costly). With a restricted number of cores, however, broadcast seems to be the most suitable choice, bearing in mind that many commercial high-performance processors use it [10][14][16]. For these systems, cost constraints support this design decision but it might not be sustainable in future designs.

One way to tackle the problem is to use suitable interconnections that minimize the utilization of the same resource in the network by copies of the same message. This will be done by supporting on-network broadcast and/or on-network gather [19][22]. At the same time, to avoid both communication and tag snoop overheads, many works advocate filtering [10][28] or adapting the protocol behavior to the bandwidth availability [25][30]. In order to filter out unnecessary memory

controller (MC) accesses or off-chip coherence fabric interfaces (XC), additional mechanisms should be provided [16][10].

C. Cache Coherence Hybridization

Many of the previous solutions proposed to alleviate both directory and broadcast limitations are based on a complementary design alternative. For example, in a directory for shared blocks, the approach followed is to snoop all or a subset of the coherence agents to see whether they have a copy of the block when a request misses in the directory and/or LLC [12][27]. In other works, the coherence protocol acts as a snoop-based protocol does. Therefore, the resulting protocol mimics a directory protocol in some cases and a broadcast protocol in others. The same observation could be made for some broadcast-based coherence protocols, which reduce the energy overheads through the insertion of structures to filter unnecessary cache snoops [28]. In some way, most of these solutions use a "base" approach (either directory or broadcast) and use the complementary one to compensate for its inherent limitations. Similarly, our proposal combines both strategies: a standard sparse directory [15] and token coherence [24]. We will use a directory-like structure to track most shared blocks (with precision in a sharing vector) and private block presence (approximately). In both cases, token coherence is used to discover when, after a miss, a block should be classified in one or other group. The information present in this structure will be used to minimize the on-chip and off-chip traffic. Then, intuitively, we can say that both facets of the coherence protocol operate with similar levels of relevance. Our objective is a mechanism able to get the best of both worlds: the performance of a broadcast-based protocol with the energy efficiency of a directory-based protocol, while incurring a minimal storage overhead.

III. FLASK COHERENCE

The coherence controller is composed of two key elements: a sparse directory that will be used only for actively shared data and a filter that will be in charge of determining when, for a given address, there is a copy of the block in any private cache. In both cases, we will use token counting [24] with two objectives: guaranteeing that *coherence invariants* are respected and monitoring when the filter should be updated. The number of tokens of a block that enters the chip is set to the number of cores. Then, a copy of a block can be read if it has at least one token. A copy of the block can be written if it has all the initial tokens. In this way, we guarantee single writer/multiple readers invariant for each block. Token counting will be useful to discover whether a block is being actively shared or not and to act accordingly, not just in the sparse directory but also in the filter. Filter management is mostly isolated from coherence protocol. In fact, the coherence controller will act correctly with an empty filter. Token counting allows us to use this strategy: it helps the filter to determine when the last copy of the block leaves or enters the chip. The filter should be understood as a traffic filtering device for both on-chip and off-chip traffic. A high-level representation of the coherence controller is shown in fig. 1. Next, we will detail how each element operates.

A. Sparse Directory (for the majority of the actively Shared Blocks)

Broadcast Token Coherence (TokenB) resolves misses in the private cache by issuing a request to all the potential coherence agents in the chip where a copy can reside. To avoid this scenario, we use a sparse-like directory. In contrast to conventional sparse directories, this will track only actively shared blocks. This means that when a block is missing in the chip, an entry will be allocated only in the corresponding private cache. If the block remains private, when it gets replaced, it will be progressively moved to further levels until it gets evicted to the LLC, which is non inclusive. We denote the time between these two points the *private caching period*. The block is actively shared if during the private caching period it is accessed by another core in the system. If this circumstance arises, we need to allocate an entry in the directory with its sharing information, i.e. with the two cores if the second request was a read, or with the last core if it was a write.

Thus when a processor misses in the corresponding private cache, it is not possible to determine whether there are copies in other private caches just by checking whether there is an entry allocated in the directory or not. Note that when a directory entry is evicted no external invalidations are sent to the private caches that hold a copy of the block. At first, after any miss in the directory, we will assume that it will initiate a reconstruction

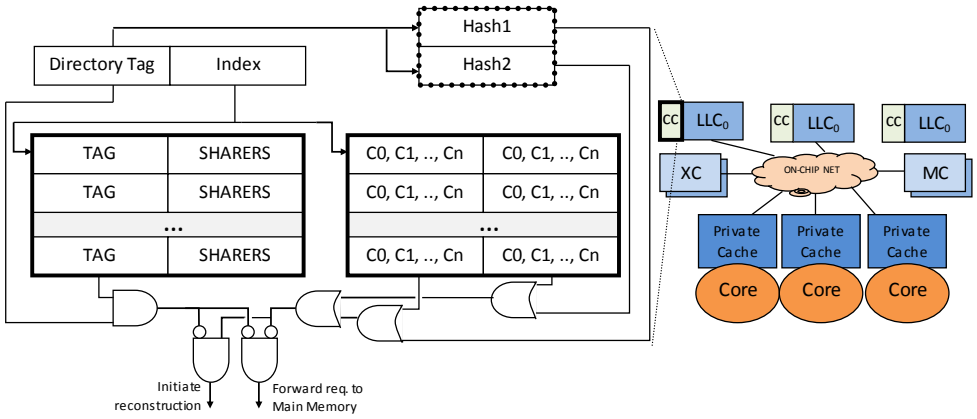

Fig. 1. High-level representation of a lookup in the FLASK coherence controller (simplified vision with a direct-mapped sparse directory and a parallel counting bloom filter with n counters and two hash functions).

process, which will snoop the remaining coherence agents in order to recreate the sharing vector. Using token counting, the directory will be aware when the information received is enough to unblock the pending memory operation. The on-chip coherence agents will respond with the count of tokens owned for the requested data.

There will be 3 potential scenarios:

(1st Case) If the response is delivered from memory, we consider that the block is private and the requesting core will be the owner of the data. Then the directory can directly forward the data and tokens to the requesting processor without allocating an entry in either the directory or the LLC (i.e. a private block will not evict the information corresponding to another shared block).

(2nd Case) If the response comes from LLC and it indicates that LLC has all the tokens, the directory knows that there is no other copy of the block in the chip and it can instruct the LLC to forward the block to the requestor processor. In this case, like in memory responses, we know that the block is not actively shared and so it will not be necessary to allocate a new entry in the directory.

(3rd Case) If any response comes from any of the cores, i.e. from their private caches, we know that the block is being actively shared and so a directory entry will be allocated. If the pending operation is a *load*, the directory instructs the private cache with the owner token to forward the data to the requestor and it keeps counting incoming answers. When the location of all the tokens is known, the sharer vector is accurate. If the pending operation is a *store*, the directory instructs all the private caches with tokens to forward them to the requestor core (consequently invalidating the data) and also requests the cache where the owner token is allocated, to forward the data block to the requestor processor. When all the tokens are received by the requesting core, it knows that it can proceed with the operation, and unblock the directory entry.

The reconstruction approach is based on the one proposed in MOSAIC [27]. Nevertheless, FLASK never allocates an entry in the directory unnecessarily (i.e. for a private block). Eventually, MOSAIC can use the directory to allocate mostly shared blocks, but on-chip misses will trigger directory replacements. However, in any case, this will imply a directory-induced private replacement like in conventional sparse directories. It will increase the effort necessary to reconstruct sharing information if the replaced entry was shared through a multicast operation. Like in the Stash Directory [12], we will allocate only shared data. In contrast to our proposal, the Stash directory requires an inclusive LLC to identify shared blocks. This means that in FLASK (or MOSAIC) more blocks missed in private caches will be located in LLC because of their greater effective capacity. In fact, the 2nd scenario will be the most common case. Requesting the block first from the LLC, in most cases, will make it unnecessary to ask the remaining coherence agents. Finally, like in MOSAIC and in contrast to the Stash directory, the eviction of an entry in the directory does not require invalidating the copies in the private caches. This will be helpful for frequent read-only blocks (such as instructions) since eventually they will be shared without using an entry in the directory. Note that many workloads, such as most server/transactional applications, have a large instruction footprint with a high sharing degree [5]. Finally, FLASK will filter unnecessary on-chip and off-chip traffic through the mechanism described next.

B. Counting Filter (for all the on-chip blocks)

Using the previously described policy to manage the directory will substantially increase the on-chip and off-chip request traffic. The second core accessing an actively shared block will be unable to determine whether it is present in the chip. Therefore, all the coherence agents should be interrogated when a miss occurs in the directory. Since our intention is to keep track of only the actively shared blocks in the directory (with the object of reducing the size), this will be the common case.

To tackle this issue, at the directory level, we have to track whether the data is present or not in the chip. Note that the aggregate private blocks tracked by this structure can be large, since both the number of cores in the chip and the private cache assigned to each one can be large. In contrast to directory, we can trade off inaccuracy in this structure in order to maintain the implementation cost constrained. This structure should also be useful to filter unnecessary memory snoops (i.e. snoop memory when there is a copy of the block present in the chip).

1) Filtering the On-chip traffic

We propose using a small counting bloom filter [6] attached to each directory entry to track all the tags contained in the private sets that map onto the entries of the directory. A counting bloom filter is an efficient, approximate set membership check that enables the tracking of a tag's presence in the private caches at a fraction of the regular cost. Since this is a filtering structure and not a correctness construction, we can tolerate the presence of false positives. These situations might increase traffic and delay memory accesses but they will not affect system correctness.

In our case, we need to increment and decrement the counters of the filter each time a block arrives or leaves the chip. The event required that triggers the increment in the counters will be an on-chip miss. This situation can be identified when the data arrives at the chip after a miss. On the contrary, decrementing each counter might be a significantly harder task. It requires identifying when there is no longer a copy of the block in the chip. We will use token counting for this purpose. As in the first scenario, if on LLC eviction a block has all the tokens, we know that there will be no other copies in the chip and consequently we can decrement the filter directly. The remaining scenario, i.e. a block that has to be evicted from LLC without all the tokens, can be much more complex to handle. Fortunately, when a block is evicted from LLC, in most cases (>99.99%) the block has all the tokens. This is an expected behavior, since the reuse distance supported by private caches is much shorter than that supported by the LLC, given the size ratios between the two caches. Instead of dealing with the problem through coherence protocol constraints, we opt to use the fact that the second case is very unlikely. Then, if we have to evict a block in LLC without all the tokens, we invalidate any private copy of the block through a broadcast. When all the tokens are collected in the LLC, we know that we can decrement the corresponding counters in the filter.

Finally, as with any counting bloom filter, the counters can overflow [4]. This situation might affect system correctness since there could be false negatives. Memory requests are likely for blocks potentially modified in the chip. The coherence protocol should contemplate this event and handle it accordingly. As this is a very unusual case, again, it is not cost effective to increase the complexity of the coherence protocol because of it. We opt for avoiding counter saturation through invalidation. If after an on-chip miss (and subsequent addition), some of the counters reach the maximum value, we invalidate all the private cache sets mapped onto that portion of the filter. Although at first sight this might seem to have a huge performance impact, in practice, with a large variety of configurations and workloads, it is extremely unlikely that this event would ever happen. This is consistent with the fact that the probability of having a counter overflow, is very unlikely [4].

2) Filtering the Off-chip traffic

Broadcast-based coherence protocols in CMP might increase the demands on the off-chip coherence agents as well as the energy overhead in unnecessary private cache tag snoops and traffic. Since this might affect a scarce resource such as the off-chip bandwidth, it is often managed by complex structures in the memory controller in charge of filtering or canceling unnecessary memory requests on the fly [10]. From our perspective, we can reuse the previously described on-chip filtering strategy to carry out this task seamlessly. When a tag is not found in the filter, as false negatives are not tolerated, we know that it will be found in memory. These are compulsory accesses. However, if there is a hit in the filter, the block will most often be in the chip, it being unnecessary to interrogate the memory. The problem we face is the false positives, which have a very low but non-zero probability. If we proceed as usual without asking the memory, the system might reach starvation. Instead of contemplating these events (e.g. through starvation detection [24]), we prevent this situation by forcing the cores without tokens to acknowledge directory entry reconstructions. In contrast to other protocols [10], the overhead of these acknowledgements is only present when there is a hit in the filter. The directory coherence controller will wait for all these answers before sending a request to the memory. Consequently, on-chip misses in which the counting bloom filter returns a false positive will be delayed until the coherence controller is aware that this block is not inside the chip. Fortunately, if the counting bloom filter is correctly dimensioned, the probability of having these false positives will be small enough to not interfere noticeably with system performance. The consequence is that memory accesses, in the worst case, will be delayed by a few tens of cycles. The positive effect of this approach is that the memory controllers do not have to be able to filter or cancel unnecessary memory requests. Note that the cost of this functionality might be substantial.

IV. Filter Design Considerations

A. Implementation: Improving Counting Bloom Filter Storage Efficiency

The storage capacity budget of the coherence controller should be used to keep block sharers in the sparse directory and an indication of the presence of a tag in any private cache in the filter. If we assume a structure similar to fig. 1, (i.e. assigning an independent filter to the private blocks that map onto the same directory entry), the storage cost of the filter and the directory is similar. Additionally, if we do not share the counters between different hashes, i.e. a parallel bloom filter, the number of read/write ports is independent of the number of hash functions. According to [34], the accuracy of this approach is close to a costlier true bloom filter.

For a conventional Counting Bloom Filter (CBF) with n private blocks mapped onto the entry, and m counters with the optimal number of hashes, according to [8], the probability of a false positive is approximately:

$$P_{fp\_CBF} \approx \left(ln(2)\right)^{m/n}$$

Then, to achieve a false positive probability of 5%, we require that $m/n>8$, which is equivalent to having at least 8 counters per private cache entry. To minimize the saturation probability, we need at least 4 bits per counter. In this case, the overflow probability is approximately $e^{-ln2}(ln\ 2)^{16}/16!)=6.8E-17$ [8]. Therefore, we need at least 32 bits per private block. This represents a saving of 50% for a 64-bit tag.

Unfortunately, conventional hash functions behave far from ideally, which unbalances counter usage in a CBF [31]. A quasi-perfect alternative is *d-left hashing* [37]. Over this base, Bonomi et al. [7] introduce a new filter called a *d-left* Counting Bloom Filter (dlCBF) that at least doubles the efficiency of the counters of a CBF, with a similar implementation cost. A high-level representation of this filter is shown in fig. 2. The filter works as follows: the table is divided into as many sub-tables as necessary (two in the example, four in practice might suffice). Each table is divided into "buckets", five in the example. Each bucket is divided into multiple cells. Each cell has a small counter and a few bits to store the address signature. The address signature (called remainder) and the bucket are computed applying a conventional hash function to the address and a permutation per

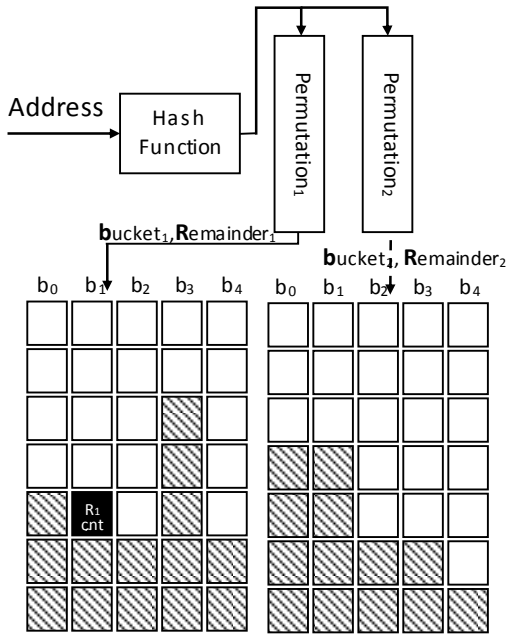

Fig. 2. Sketch of a dlCBF filter. Each cell stores a remainder and a counter.

sub-table to the previous result. Both can be done using simple combinatory logic [34]. The counter is only employed for the unusual case of different addresses with the same remainder (in practice 2 or 3 bits suffice).

When we compute the bucket for each sub-table, we allocate one entry in the least used bucket. The key element in this filter is *d-left* hashing, which establishes that if two destination buckets for an address have the same number of entries, the left one should always be chosen [37]. This allows near-perfect cell load usage, and consequently better coverage for the filter. In the case of the example, the bucket chosen is b_1 for the first sub-table and b_2 for the second one. If we assume that both have the same number of entries allocated (two different remainders with the same number of occurrences), we choose the left table to add the new remainder. If we assume that the new remainder is not present in the table, we allocate a new entry for it and set the cell counter to 1. To look up the filter, the operation is similar. Note that the total number of bits per bucket is pretty small, so the search can be done in an external register avoiding the need for a content addressable memory. Since all sub-tables are handled independently, like in a parallel bloom filter, we can use a conventional single-ported SRAM memory. Therefore, like in a conventional CBF, we can use the same storage to keep either directory or filter information. The controller logic should route each access in accordance with its use. Note that filter updates are done on LLC replacement or on-chip misses; in the first case, outside the critical path, and in the second one, overlapped with main memory access. Since the filters are banked, even in the case of having thousands of pending on-chip memory operations, the number of pending operations per filter will be low. Consequently, we can assume that the cost of updating the filter will be negligible. The look-up of a filter with A sub-tables is similar to a conventional parallel counting bloom filter with A hash functions: each bucket in the corresponding sub-table is read in parallel. We should look up the presence of the remainder in the bucket. This is equivalent to a tag search in an A-associative cache, we assume that the look-up operation is the same as that required for an equivalent directory.

For a dlCBF, the probability of false positives, with a d sub-table configuration of b buckets each, and r bits per remainder is given by [7]:

$$P_{fp\_dlCBF} = 1 - (1 - \frac{1}{d \cdot b \cdot 2^{-r}})^n \approx \frac{n}{d \cdot b} 2^{-r}$$

Assuming c cells per bucket and a utilization factor of ρ (<1), and choosing a total number of buckets depending on n as $n/(\rho \cdot c \cdot d)$:

$$P_{fp\_dlCBF} \approx \rho \cdot c \cdot d \cdot 2^{-r}$$

Thus, assuming four sub-tables (d=4), eight cells per bucket (c=8) and a 3/4 usage per bucket (ρ=0.75), to achieve a 5% false positive probability, the number of bits per remainder is 9. Assuming a 3-bit counter, each cell requires 12 bits. To compensate for bucket utilization, we have to multiply this number by 1/ρ ~16 bits which is necessary per element tracked. This, is less than half the number required by the conventional bloom filter mentioned above.

In practice, for a 64-bit physical tag (which might include sharing vector and block state), this will lead to a 77% saving. For a 256KB L2 and 32 KB L1I, 32 KB L1D and block sizes of 64 bytes, we will require ~9.2KB per core (less than 3% of the tracked cache size), whereas a conventional CBF will require about 20KB. For this configuration each bucket uses 128 bits, which simplifies table lookup. If we reduce the filter size (to save area), the false positive probability might increase, but not in a significant way. Note that all the theoretical estimations are made assuming no spatial locality or sharing in the stream of addresses.

In regard to the overflow probability of the 2-bit counter, a higher bound for the configuration assumed with 5k blocks per private cache, with *cnt=3* bits per counter, is given by [7]:

$$\binom{n}{2^{cnt}+1}\left(\frac{1}{\#buckets \cdot 2^r}\right)^{2^{cnt}+1} \approx \binom{5000}{9}\left(\frac{1}{256 \cdot 2^9}\right)^9 \approx 4.6 \cdot 10^{-19}$$

In any case, we have provided a fallback mechanism to rescue the system in this situation. Note that the system could be running continuously for months or years, and so the inclusion of this corner case is not an option.

B. Filter/Directory Resource Partitioning

The filter and the sparse directory are "complementary" structures, since actively shared blocks will reduce the load on the filter and private blocks will never use directory entries. Initially, we will assume storage resources statically splitting the available SRAM for one or other purpose. Restricting the filter capacity will increase the probability of false positives, which will increase the on-chip traffic and delay off-chip misses. Restricting the directory capacity to the point that the working set of the actively shared blocks does not fit will increase the reconstruction probability. This could be used effectively to provide a low tracking capability (perhaps for less than 20% of private cache blocks).

dlCBF is able to reconstruct the members from the remainder and bucket information. To do so, the permutation used to fill up the tables has to be invertible [7]. Then, it is possible to reconstruct a sub-table by observing the remainder sub-table and simply rolling back the permutation. Thus, it is possible to "move" the tracking of a block from one sub-table to another. In this way, if we detect that the directory is highly loaded (through the frequency of reconstructions), we can adaptively de-allocate one of the entries in the filter and expand the number of ways in the directory. Similarly, if the workload is not using the directory entries because all/most of the data are private, we can expand the filter with additional sub-tables, reducing the false positives without increasing the reconstruction frequency. Note that this operation is not possible in a conventional CBF.

Initially, for all workloads, we will equally split the storage capacity between the filter and the directory. Later, and after observing the application's behavior, we will explore workload-dependent partitioning in order to see how beneficial this approach can be in reducing adverse effects in extreme cases. We will not discuss implementation cost, although it does not seem to be an issue. Perhaps some multiplexors will be necessary to connect the SRAM content to the coherence controller or the filter. In any case, note that the dynamic

behavior will have a negligible impact on performance since the one-by-one migration of filter sub-table entries to directory could be done in the background with the normal system operation. The opposite operation is direct (just requires invalidating directory entries that will be used as a sub-table). We will not discuss the mechanism for triggering this process since the sharing pattern of the applications is quite stable throughout the execution. The most cost effective way is to trigger it by software at the beginning of the workload.

V. EVALUATION METHODOLOGY

A. System Configuration

To analyze FLASK, we model a CMP with out-of-order cores that mimics the execution resources and on-chip cache hierarchy of the Intel Haswell processor [16]: using 6-wide issue cores with 196 in-flight instructions and up to 64 pending memory operations. The number of cores in the CMP is 16. Therefore, the coherence fabric has to support up to 1024 concurrent memory operations. There are three levels of cache. The first two are private, strictly non-inclusive (i.e. L2 acts as a victim cache of L1). The third level is shared and uses a mesh network, which is characterized by better on-chip bandwidth scalability than a ring network. We will assume that the routers in the network can handle multicast traffic natively [19], have single-cycle low-load pass-through [23], separate virtual-networks to avoid end-to-end protocol deadlock and over-provisioned buffering (90 flits per port). Similarly to the LLC, the directory is banked and interleaved by the least significant bits. To quantify FLASK's properties and to understand how it behaves compared to snoop and directory-based protocols, we have implemented two reference protocols based on TokenB [24] and on a sparse-directory [15] respectively. TokenB has been selected because it allows a scalable out-of-order network to be used without adding additional mechanisms outside the coherence controllers. Using the same methodology and tools, all coherence protocols have been optimized fairly. A full SLICC specification for a 3-level hierarchy can be found in [38]. Memory bandwidth is over-provisioned to avoid the necessity of tuning memory controller architecture to each protocol. In practice, contention is negligible in all protocols. In a more realistic environment, broadcast protocol should be handled carefully to avoid unnecessary memory requests wasting a very scarce resource such as off-chip bandwidth [10].A summary of the other main parameters used in our analysis is shown in Table I.

B. Workloads & Simulation Stack

We will use GEMS [26] as the main tool for our evaluation. With GEMS, it is possible to perform full-system simulations. Coherence protocols have been implemented using the SLICC language (Specification Language for Implementing Cache Coherence). In order to model accurately interconnection network contention and its impact on the average access time, we replace the original network with TOPAZ [1]. For power and cost modeling, we use CACTI 6.5 [29] for the cache and DSENT [36] for the network. Ten workloads, shown in Table II, are considered in this study, including both multi-programmed and multi-threaded applications (scientific and server) running on top of the Solaris 10 OS. The numerical applications are three of the NAS Parallel Benchmarks suite (*OpenMP* implementation version 3.4 [20]). The server benchmarks correspond to the whole Wisconsin Commercial Workload suite [2]. The remaining class corresponds to multi-programmed workloads using part of the SPEC CPU2006 suite [35] running in rate mode (where one core is reserved to run OS services). The mix of workloads has been selected trying to cover diverse usage scenarios, varying the sharing degree (from none in SPEC applications to a large amount in Server Workloads) and sharing contention (from none in SPEC to a large amount in scientific applications). Among the NAS applications, we chose the three with the highest sharing contention. From the SPEC suite, we chose three applications with a variable range in working set size. We should emphasize that the three families of applications exhibit quite dissimilar behavior from the coherence protocol perspective, but they have to be considered, given the usage scenarios of general purpose CMPs. Focusing the evaluation on a single suite of benchmarks, it is hard to consider all the characteristics we have represented with the selected mix.

We model hardware-assisted TLB fill and register window exceptions for all target machines. Multiple runs are used to fulfill strict 95% confidence intervals (error bars are not visible in most cases). Benchmarks are fast-forwarded to the point of interest, during which page tables, TLBs, predictors, and caches are warmed up. In iteration-based applications, such as NPB, a

TABLE I. SUMMARY OF 16-CORE CMP SYSTEM CONFIGURATION

Core Arch.	Functional Units	4xI-ALU/4xFP-ALU/ 4xD-MEM
	ROB size / Issue Width	196, 6-way
	Frequency, count	3Ghz, 16 cores
Private Caches	(L1)Size/Associativity / Block Size / Access Time/Repl.	32KB I/D, 4-way, 64B, 1 cycle, LRU
	(L2)Size / Associativity/ Block Size / Access Time/Repl.	256KB Unified, 8-way, 64B, 2 cycles, LRU, Exclusive with L1
	Outstanding Requests per core	64
Shared L3	Size / Associativity / Block Size/Repl.	16MB (or 32MB), 16 (or 32)×1MB, 16-way, 64B , LRU
	NUCA Mapping	Static, interleaved by LSB
	Slice Access Time	6 cycles
Mem	Capacity / Access Time / Memory Controllers / BW	4GB, 240 cycles, 4/32GBs
Network	Topology / Link Latency / Link Width/Clock	4×4 Mesh, 1 cycle, 16B, 3Ghz
	Router Latency (low-load) / Flow Control / Routing/ Buffering	1-3 cycle/ Wormhole/DOR/10KB

TABLE II. MULTITHREADED WORKLOADS

SERVER [2]	OLTP	IBM DB2 DBMS, TPC-C like 10000 Transactions
	Apache	Apache web server, SpecWeb like, 25000 Transactions
	JBB	SpecJBB, 70000 Transactions
	Zeus	Zeus web server, SpecWeb like, 25000 Transactions
NPB[20]	Multi-Grid (MG)	CLASS A
	Fast Fourier Transform (FT)	CLASS W
	LU Diagonalization (LU)	CLASS A
SPEC [35]	Astar	Native, 15 thr.
	Hmmer	Native, 15 thr.
	Omnetpp	Native, 15 thr.

warm checkpoint is taken in the middle of the execution and with a reduced number of iteration runs. Transactional workloads are warmed up by running hundreds of thousands of transactions, and accurately simulated for a fixed number of transactions. SPEC workloads are fast-forwarded to the point of interest and simulate ~8billion instructions.

VI. Performance Results

A. Comparative results with reference protocols

The fundamental parameters of the system for the considered coherence protocols running the selected workloads, namely *execution time*, *average access-time* and memory hierarchy *energy delay product* (EDP) are shown from fig. 3 to fig. 5. To clarify the total storage, we denote it as SDE (*Sparse-Dir Equivalent*) capacity. All the results are normalized against token coherence and the directory size is being swept from a capacity to track 160% of the private cache blocks (8K SDE entries) to just 5% (32 SDE entries). Note that in order to keep implementation cost constant, in FLASK, at this point, half of this capacity is devoted to the filter and half to the directory. In other words, in the most extreme case, the FLASK directory will only have SDE capacity to track 16 shared blocks per controller (256 in the whole chip).

As expected, when the size of the directory is below one fourth of the aggregate private caches' capacity, the sparse-directory performance is degraded. In contrast to other previous works, [12], we observe this degradation with significant directory size reduction. This is related to the fact that in that work there is only one level of private caches. Here, L2 acts as a victim cache for L1. Thus, a directory induced private miss is eight times more frequent in L2 than L1. If we take into account that in L2, block reuse is low [18], the results seem reasonable. With an inclusive L1/L2 (which for an aggressive out-of-order core and a shared LLC might not be interesting), the effects will be more noticeable but still not too acute. Additionally, it should be noted that for an in-order core, directory-induced private misses (and subsequent hits in L3) will have more effect on performance. The memory level of parallelism present in the evaluated system allows the impact to be partially hidden.

As can be seen in fig. 5, sparse directory hits in private caches are reduced when the count of tracked blocks is decreased. Thus, there is a substantial increment as data must be

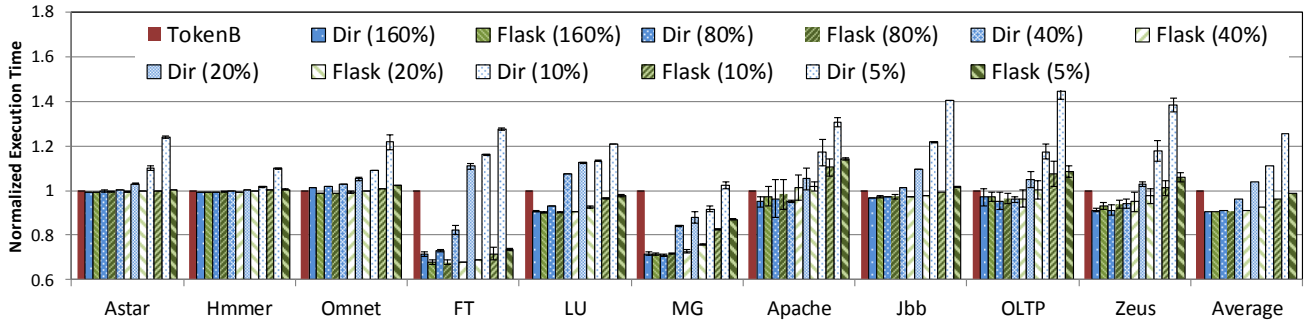

Fig. 3. TokenB Normalized Execution Time.

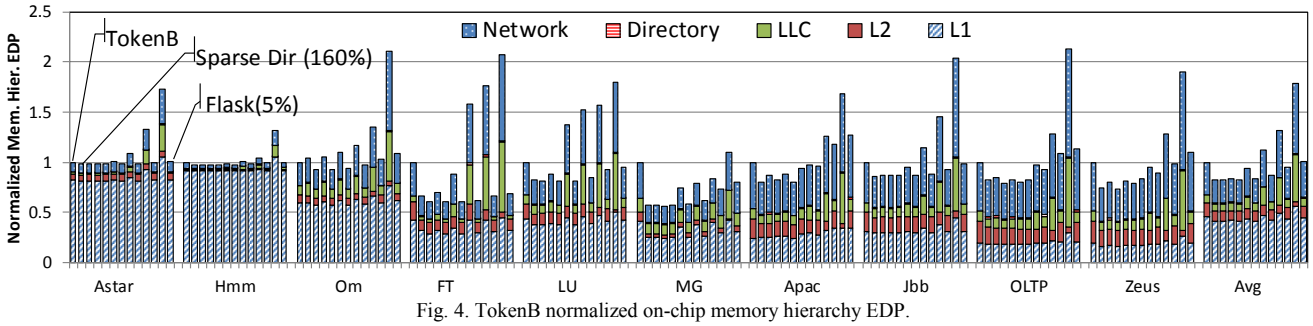

Fig. 4. TokenB normalized on-chip memory hierarchy EDP.

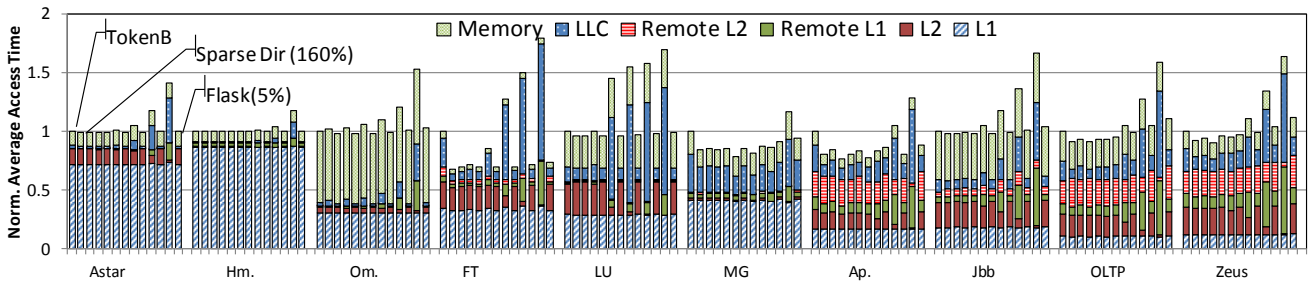

Fig. 5. TokenB normalized average memory access time.

retrieved from LLC (for private blocks) or other private caches (for actively shared blocks). In numerical or multi-programmed workloads the former case is more frequent while the latter exists in server workloads. This is consistent with the sharing degree. As can be seen for applications with a large portion of shared data, the latency degradation from 160% to 5% is almost doubled, degrading the performance by more than 40% on average. For these cases the intense coherence traffic due to directory-induced invalidations and subsequent LLC hits makes the activity increase in the network substantially, which degrades the energy properties, making it worse than the snoop-based protocol.

When we look at the performance of the snoop-based coherence protocol, we can observe unstable behavior. In some applications it seems to be almost the best performer while in others it is clearly not. The reason can be found in the on-chip contention. For some of the scientific applications, there is a significant traffic pressure on the LLC network. In spite of having a reasonably dimensioned network (4x4 mesh with 3 GHz clock and 16Byte links) and a router with state-of-the-art features [19] [23], the latency of the LLC is larger. Despite being an average sized system, extra traffic imposed by broadcast requests and the concurrent memory operations (at a given time there may be more than 16K packets in-flight) seem to surpass network capabilities. The solution for this unpredictable behavior is to oversize the network (e.g. increasing the link width, using topologies with better connectivity, using a more advanced router, etc.) or to redesign the coherence protocol [30]. Unfortunately this would be inappropriate for bigger systems. In any case, even in applications where TokenB has a slight performance advantage, the energy consumption is higher.

FLASK exhibits very different behavior to the other protocols. Even in the most extreme configuration, the performance seems quite unaffected. In the worst case, which is *apache*, the performance degradation observed versus token is 13%. In general, it seems that applications with low sharing degree are correctly handled, with almost negligible performance degradation. Having only SDE capacity to track 256 private blocks seems to affect other applications more significantly. This causes more reconstructions, which delays access to shared data after a capacity miss in private caches, slightly lengthening LLC access time. In other applications, such as the multi-programmed workloads, in spite of having a minimal filter (just 32 buckets per sub-table), the effects both on performance and energy of this extreme configuration are negligible. On average the performance degradation when reducing from 160% SDE capacity to 5% is only 8%, which is a noteworthy result. In all cases, the number of private-cache external invalidations due to the replacement of blocks in LLC without all the tokens is negligible.

Since the on-chip traffic is filtered out, there is no similar behavior to that observed for TokenB. In general, the energy requirements of the protocol are smaller than token and are quite steady regardless of the directory size. Note that this metric is pessimistic, since we exclude the cores' power consumption. As FLASK is the best performer in most cases, the EDP of these components will be low.

In summary, with almost no tracking capability, FLASK is more stable and energy efficient than broadcast-based protocol, and its performance degrades much more gracefully than conventional sparse-directory when directory size is reduced.

B. *Filter Efficiency*

The previous results contain some details that we want to highlight, namely, the filter efficiency. The main figure of merit is the false positives, which increase on-chip energy and memory

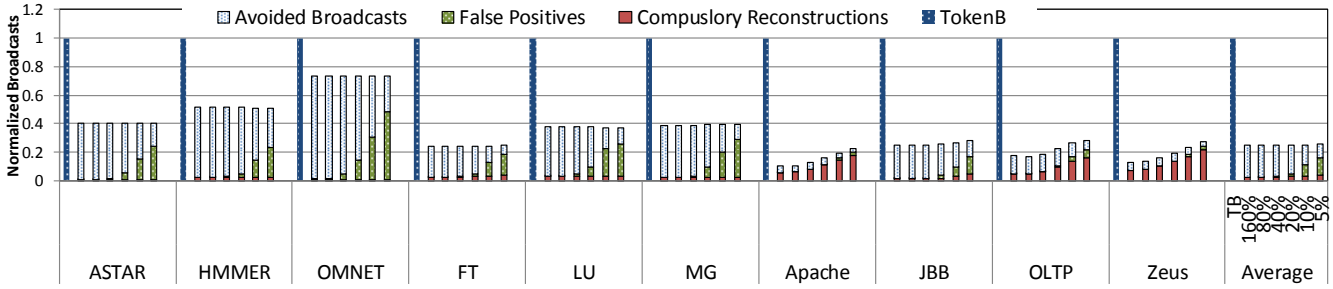

Fig. 6. TokenB normalized filter efficiency for different SDE sizes.

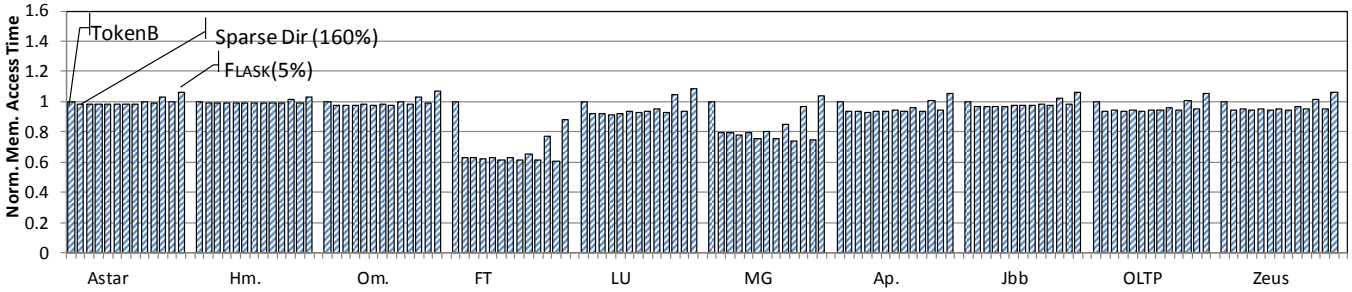

Fig. 7. TokenB normalized memory latency overhead.

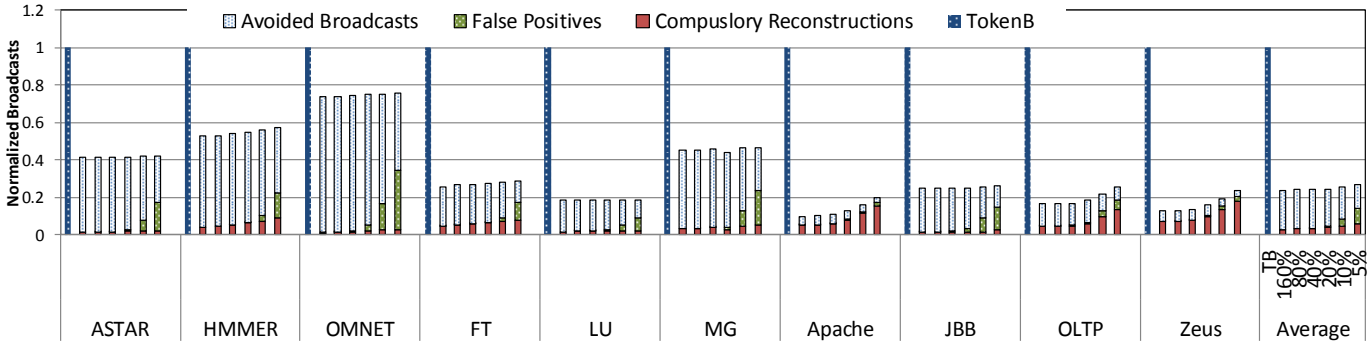

Fig. 8. TokenB normalized traffic filtering with adaptive partitioning.

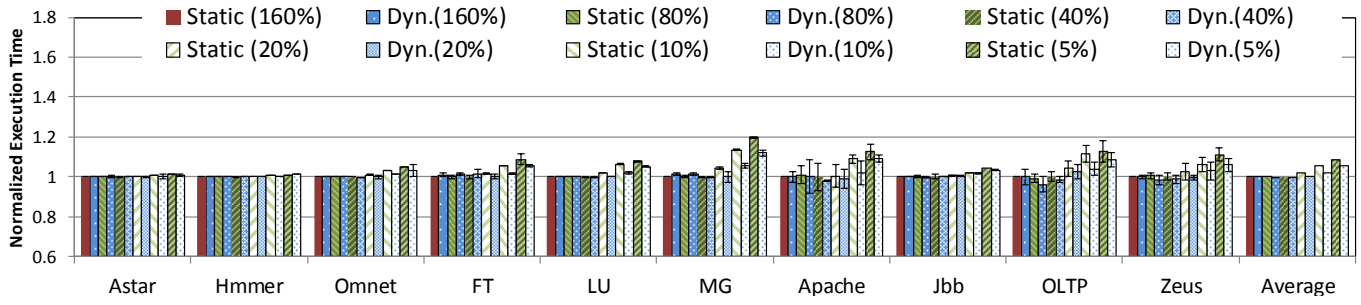

Fig. 9. 40% SDE Static normalized performance of FLASK with dynamic and static storage allocation.

access time. Fig. 6 shows the total number of multicasts over a range of available storage for the filter capacity.

For reference, we normalize this metric to multicast in TokenB. Note that, when we take into account that the network can natively handle multicast traffic and the cache snoop energy contribution, this might not be directly transferred to link utilization or energy consumption, as we can contrast these results with fig. 4. When the directory is dimensioned for 20% of private caches, false positives seem to be consistent with the theoretically expected proportion of 5%. If we shrink the SDE capacity to just 5% of private caches, although there is an increase in false positives, counter saturation never occurs in any of the runs of our evaluations. While in some cases, even with such a small filter, it is able to detect a reasonable number of on-chip misses; in other cases (such as the NAS applications), it cannot do so. In some applications, the number of true positives (private cache misses for actively shared data) grows when we reduce the size of the directory while in others, it remains almost unchanged. In the former, the directory is not able to maintain the shared working set, whereas in others it can do so. In contrast to this behavior, false positives are more numerous in the latter applications as private blocks increase the number of elements to be tracked by the filter.

Fig. 7 shows the TokenB normalized memory access time (from the core perspective, i.e. includes the whole latency to transmit the desired word from memory to the processor back-end) for FLASK and sparse-dir. Note that memory requests in both cases will be roughly the same, since most off-chip requests are induced by LLC capacity misses and both protocols handle LLC data in the same way. In numerical applications, false positives for very small filter capacities have a negative effect due to the delay in off-chip access and the on-chip latency that the extra traffic adds. Consequently, there is a significant increase in memory access time. In applications such as the server workloads, the extra traffic due to compulsory reconstructions increases the contention, increasing the access time to the memory controller and therefore delaying the memory access.

For the generous off-chip bandwidth considered, the results are dominated by on-chip effects: on-chip contention and the additional latency induced by false positives (which delays the memory request until the coherence controller realizes that there is no on-chip copy). Even in the most adverse directory configuration, the effect is less than 5%. This is almost unnoticeable in the average access time, as can be appreciated in fig. 5. Note that in a system with many cores or large private caches, this configuration could reduce the storage footprint significantly. In any case, directory size can be a useful knob for the trade-off between implementation and energy cost.

C. *Adaptive Filter/Directory resource partitioning*

As stated before, dlCBF allows adaptive resource splitting. Although we did not implement the on-line adaptation, we can statically choose the best configuration for the application. Looking at fig. 6, it seems clear that multi-programmed, numerical applications require few entries in the directory. On the contrary, for commercial workloads, the on-chip traffic is dominated by reconstructions. Therefore, we should select only a 1-way directory for the first two classes (with 7 sub-tables in the filter) and a 7-way directory for the last class (with just 1 sub-table in the filter). Note that even in a multi-programmed workload, a small number of OS addresses might be shared, so we need at least a 1-way directory. The broadcast signature, normalized against the token, is shown in fig. 8. The

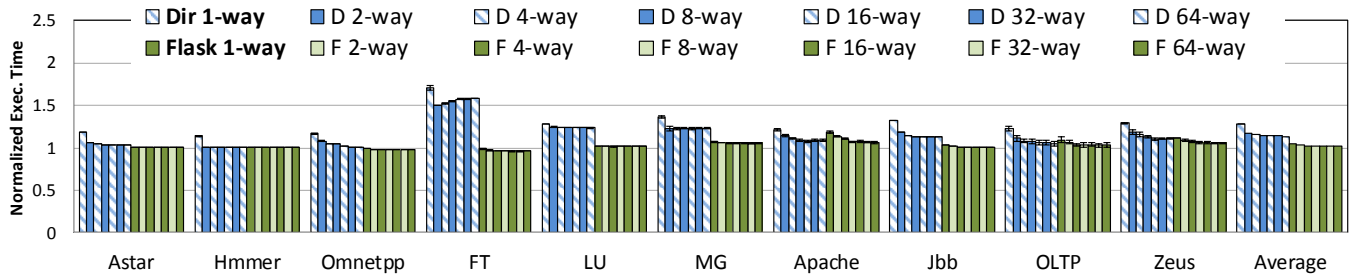

Fig. 10. 160% SDE sparse directory normalized performance for different associativities (20% SDE).

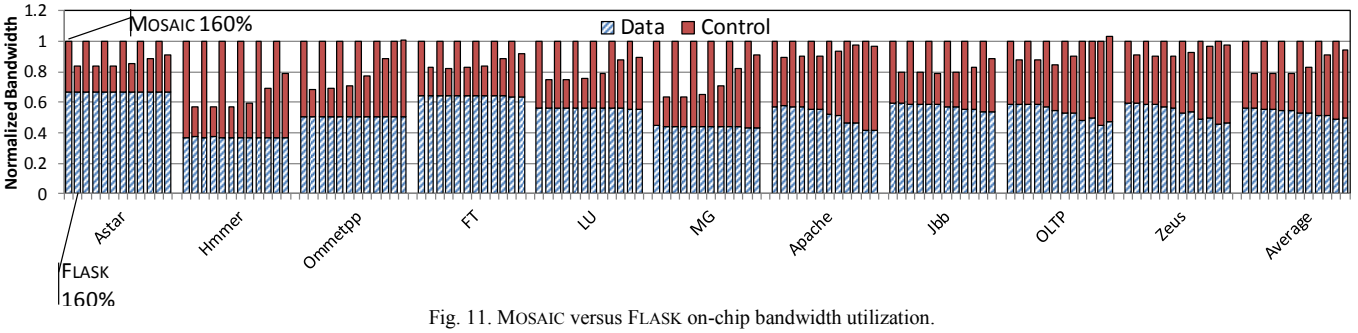

Fig. 11. Mosaic versus Flask on-chip bandwidth utilization.

performance results of the static and dynamic storage allocation in the memory controller are shown in fig. 9. If we compare these results with those provided in fig. 6, the effect of the approach is to halve the storage resources with negligible impact. For example, under these conditions with 10% SDE capacity, the traffic requirements are close to 20% of SDE capacity when the results are statically halved. Consequently with just 10%, in most applications, a tolerable false positive rate is observed. On average, with 10% and 5% of SDE capacity, we observe a performance penalty of about 3% and 6% respectively. Although dlCBF is suitable for an on-line mechanism, capable of morphing sparse-directory and filter throughout the application execution, we left this analysis open for future work. Under cloud-computing scenarios (with live workload migration), this might be interesting, given the results seen here. However, in any case, we believe that such a task should be done in the software layer, since the switch in behavior will be quite infrequent.

D. Comparing Flask with other directory cost reduction alternatives

Flask can be combined with other strategies focused on the same goal [11][12][13][27][32]. Some of them are focused on eliminating the directory overprovision by emulating large associativity through multi-hashing indexing and insertion [32] [33], achieving in most workloads the benefits of a very high associativity at a fraction of the cost. Therefore, if we increase the associativity of the sparse directory, we can achieve a similar effect. Fig. 10 shows the over- provisioned normalized result for different associativities (ranging from 1-way to 64-way with a fixed capacity). In order to appreciate conflict misses in the directory, and given that the computational requirements of the evaluation framework system and application scalability prohibit it, we artificially reduce the size to just 20% SDE (otherwise evictions caused by conflict in the directory are negligible). Except in the case of FT, whose counterintuitive behavior is caused by very low directory reuse [32], increasing the associativity reduces the directory conflicts, which provides a slight performance degradation. As we can appreciate, 1-way directory in Flask is able to outperform the sparse directory even with 64 ways. In Flask, the impact of directory conflicts on performance is negligible. Therefore, Flask will provide better performance than techniques focused on minimizing directory conflicts such as [11][12]. The reasons for this are: (1) there is no need to perform external invalidations after a directory eviction, and (2) it only uses the directory to track actively shared blocks. Under such circumstances, the experimental observations made by [15] about conflicts are no longer applicable. In fact, although not exploited here, Flask can be used to reduce directory implementation cost (v.gr. using very low associativity).

Like Flask, Mosaic was focused on improving directory scalability, eluding directory inclusiveness through entry reconstruction on demand. Fig. 11 compares the on-chip memory bandwidth consumption for the two approaches, for different SDE capacities. As expected, the control traffic generated by Flask is significantly less than Mosaic. This is because directory entry reconstructions in Flask are performed only for shared data. This means that in some multi-programmed workloads, such as *hmmer,* the total amount of traffic is up to 60% less. When the size of the filter is reduced, the false positives increase the traffic slightly. With applications with a high sharing degree, such as *apache,* this advantage is smaller. In this last type of benchmarks, when the size of directory is reduced below 20%, the behavior of the two protocols becomes more similar due to the fact that most reconstructions are because of shared blocks. On average, and with a directory of at least 20%, Flask enables the reduction of total on-chip traffic by 20%.

VII. CONCLUSIONS

We have proposed an evolutionary re-architecture for directory coherence protocols that can benefit from snoop-based coherence without paying a high toll. The results enable a balanced approach that improves system performance in a wide range of applications.

The proposal might be beneficial to scale the coherence protocol for many-core systems or for medium-size CMPs, as we have demonstrated in the results section especially so bearing in mind recent and forthcoming commercial systems. In any case, we have shown that even for 16-core CMPs there are both power and performance benefits versus other protocols.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their helpful comments. Special thanks to Dan Sorin and Viji Srinivasan for their valuable suggestions.

REFERENCES

[1] P. Abad et al., "TOPAZ: An Open-Source Interconnection Network Simulator for Chip Multiprocessors and Supercomputers," in *Int. Symposium on Networks-on-Chip (NOCS)*, 2012, pp. 99–106.

[2] A. R. Alameldeen et al., "Simulating a $2M commercial server on a $2K PC," *Computer (Long. Beach. Calif).*, vol. 36, no. 2, pp. 50–57, Feb. 2003.

[3] M. Alisafaee, "Spatiotemporal Coherence Tracking," in *Int. Symposium on Microarchitecture (MICRO)*, 2012, pp. 341–350.

[4] J. Almeida and A. Z. Broder, "Summary cache: a scalable wide-area Web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[5] I. Atta, P. Tözün, X. Tong, A. Ailamaki, and A. Moshovos, "STREX," in *Int. Symp. on Computer Architecture (ISCA)*, 2013, vol. 41, no. 3, p. 273.

[6] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[7] F. Bonomi, M. Mitzenmacher, and R. Panigrahy, "An improved construction for counting bloom filters," in *ESA'06 14th Annual European Symposium*, 2006, pp. 684–695.

[8] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Math.*, vol. 1, no. 4, pp. 485–509, Jan. 2004.

[9] M. Butler, "AMD 'Bulldozer' Core - a new approach to multithreaded compute performance for maximum efficiency and throughput," in *Symposium on High-Performance Chips (HotChips)*, 2010.

[10] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, Mar. 2010.

[11] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *Int. Symp. on Computer Architecture (ISCA)*, 2011, p. 93.

[12] S. Demetriades and S. Cho, "Stash Directory: A Scalable Directory for Many-Core Coherence," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

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

[14] E. J. Fluhr et al., "POWER8: A 12-core server-class processor in 22nm SOI with 7.6Tb/s off-chip bandwidth," in *International Solid-State Circuits Conference (ISSCC)*, 2014, pp. 96–97.

[15] A. Gupta, W. Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *Int. Conference on Parallel Processing (ICPP)*, 1990, pp. 312–321.

[16] P. Hammarlund et al., "Haswell: The Fourth-Generation Intel Core Processor," *IEEE Micro*, vol. 34, no. 2, pp. 6–20, 2014.

[17] J. Huh et al., "A NUCA substrate for flexible CMP cache sharing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 8, pp. 1028–1040, 2007.

[18] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Int. Symposium on Computer Architecture (ISCA)*, 2010, pp. 60–72.

[19] N. E. Jerger, L. S. Peh, and M. Lipasti, "Virtual circuit tree multicasting: A case for on-chip hardware multicast support," in *International Symposium on Computer Architecture (ISCA)*, 2008, pp. 229–240.

[20] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," *NASA Ames Research Center," Technical Report NAS-99-011*, Citeseer, 1999.

[21] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, Mar. 2005.

[22] T. Krishna, L.-S. Peh, B. M. Beckmann, and S. K. Reinhardt, "Towards the ideal on-chip fabric for 1-to-many and many-to-1 communication," in *Int. Symposium on Microarchitecture (MICRO)*, 2011, vol. 2, pp. 71–80.

[23] A. Kumary, P. Kunduz, A. P. Singhx, L.-S. Pehy, and N. K. Jhay, "A 4.6Tbits/s 3.6GHz single-cycle NoC router with a novel switch allocator in 65nm CMOS," in *International Conference on Computer Design (CCD)*, 2007, pp. 63–70.

[24] M. M. K. Martin, M. D. Hill, and D. A. Wood, "Token Coherence: decoupling performance and correctness," in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, 2003, pp. 182–193.

[25] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood, "Bandwidth adaptive snooping," in *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 251–262.

[26] M. M. K. Martin et al., "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Comput. Archit. News*, vol. 33, no. 4, p. 92, Nov. 2005.

[27] L. G. Menezo, V. Puente, and J. A. Gregorio, "The case for a scalable coherence protocol for complex on-chip cache hierarchies in many-core systems," in *Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013, pp. 279–288.

[28] A. Moshovos, "RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence," in *Int. Symp. on Computer Architecture (ISCA)*, 2005, pp. 234–245.

[29] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *Int. Symposium on Microarchitecture (MICRO)*, 2007, pp. 3–14.

[30] A. Raghavan, C. Blundell, and M. M. K. Martin, "Token tenure: PATCHing token counting using directory-based cache coherence," in *Int. Symposium on Microarchitecture (MICRO)*, 2008, pp. 47–58.

[31] M. V. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *IEEE Trans. Comput.*, vol. 46, no. 12, pp. 1378–1381, 1997.

[32] D. Sanchez and C. Kozyrakis, "SCD: A scalable coherence directory with flexible sharer set encoding," in *18th IEEE Int. Symposium on High Performance Computer Architecture*, 2012, pp. 1–12.

[33] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling Ways and Associativity," in *MIcro 2010*, 2010, pp. 187–198.

[34] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam, "Implementing Signatures for Transactional Memory," *Int. Symp. Microarchitecture*, pp. 123–133, 2007.

[35] SPEC Standard Performance Evaluation Corporation, "SPEC 2006." .

[36] C. Sun et al., "DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling," in *Int. Symposium on Networks-on-Chip (NOCS)*, 2012, pp. 201–210.

[37] B. Vöcking, "How asymmetry helps load balancing," *J. ACM*, vol. 50, no. 4, pp. 568–589, Jul. 2003.

[38] "SLICC specification of Flask and Counterpart Coherence Protocols." [Online]. Available: http://www.atc.unican.es/galerna/flask.