# *In-network coherence filtering: Snoopy coherence without broadcasts*

# In-Network Coherence Filtering: Snoopy Coherence without Broadcasts

‡Niket Agarwal, ⋆Li-Shiuan Peh, and ‡Niraj K. Jha
‡Department of Electrical Engineering, Princeton University
⋆Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology
{niketa@princeton.edu, peh@csail.mit.edu, jha@princeton.edu}

## ABSTRACT

With transistor miniaturization leading to an abundance of on-chip resources and uniprocessor designs providing diminishing returns, the industry has moved beyond single-core microprocessors and embraced the many-core wave. Scalable cache coherence protocol implementations are necessary to allow fast sharing of data among various cores and drive the many-core revolution forward. Snoopy coherence protocols, if realizable, have the desirable property of having low storage overhead and not adding indirection delay to cache-to-cache accesses. There are various proposals, like Token Coherence (TokenB), Uncorq, Intel QPI, INSO and Timestamp Snooping, that tackle the ordering of requests in snoopy protocols and make them realizable on unordered networks. However, snoopy protocols still have the broadcast overhead because each coherence request goes to all cores in the system. This has substantial network bandwidth and power implications. In this work, we propose embedding small in-network coherence filters inside on-chip routers that dynamically track sharing patterns among various cores. This sharing information is used to filter away redundant snoop requests that are traveling towards unshared cores. Filtering these useless messages saves network bandwidth and power and makes snoopy protocols on many-core systems truly scalable. Our in-network coherence filters are able to reduce the total number of snoops in the system on an average by 41.9%, thereby reducing total network traffic by 25.4% on 16-processor chip multiprocessor (CMP) systems running parallel applications. For 64-processor CMP systems, our filtering technique on an average achieves 46.5% reduction in total number of snoops that ends up reducing the total network traffic by 27.3%, on an average.

## Categories and Subject Descriptors

C.1.2 [**Computer Systems Organization**]: Multiprocessors—*Cache coherence protocols, Interconnection architectures*

## General Terms

Design, Measurement, Performance

## 1. INTRODUCTION

With continued transistor scaling providing chip designers with billions of transistors, architects have embraced many-core architectures to deal with increasing design complexity and power consumption [1,2,13,27]. In these systems, a shared memory model eases programmability. Hence, as core counts increase, a scalable cache coherence implementation is critically needed.

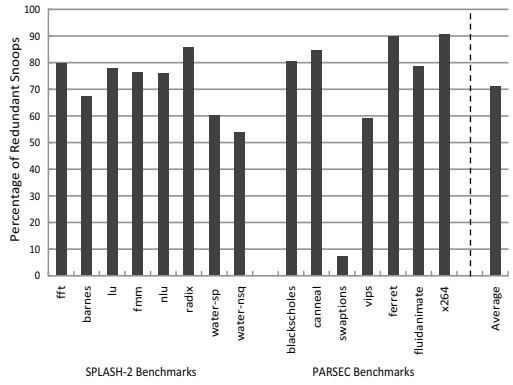
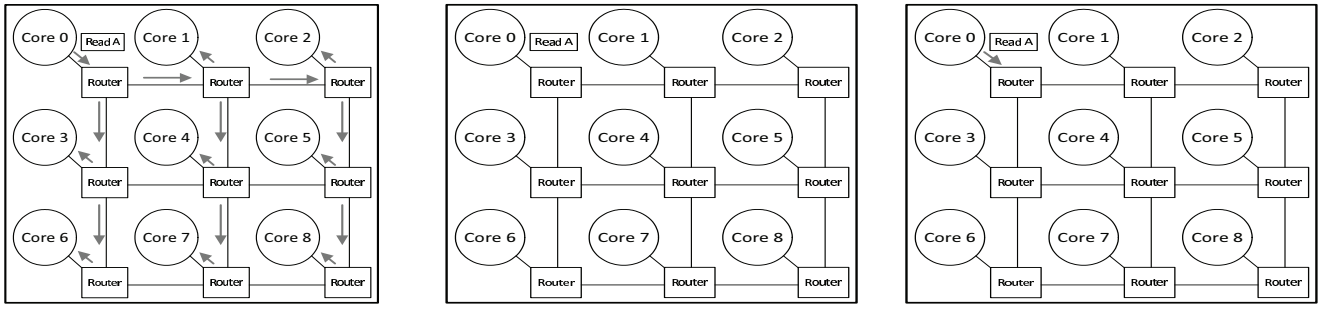



**Figure 1: Redundant snoop requests in parallel applications.**

There are two broad sets of cache coherence protocols that have been proposed: broadcast-based snoopy protocols and directory-based protocols. Both sets of protocols have their advantages and disadvantages. Broadcast-based snoopy coherence protocols [4,15,17,18,24] have the advantage of direct cache-to-cache transfers (1-hop protocol) and do not require a directory structure. However, the main limitation of these protocols is that they rely on broadcasting cache misses, which places exorbitant demands on on-chip network bandwidth beyond a moderate number of cores. The directory protocols get rid of this problem of broadcasting coherence requests by maintaining sharer information at distributed directory nodes. However, the traversal to the directory node and directory look-up introduces indirection (2-hop protocol) in cache-to-cache accesses. Directory protocols also have to store the directory structure on-chip, which introduces storage overheads. These overheads worsen as core count increases.

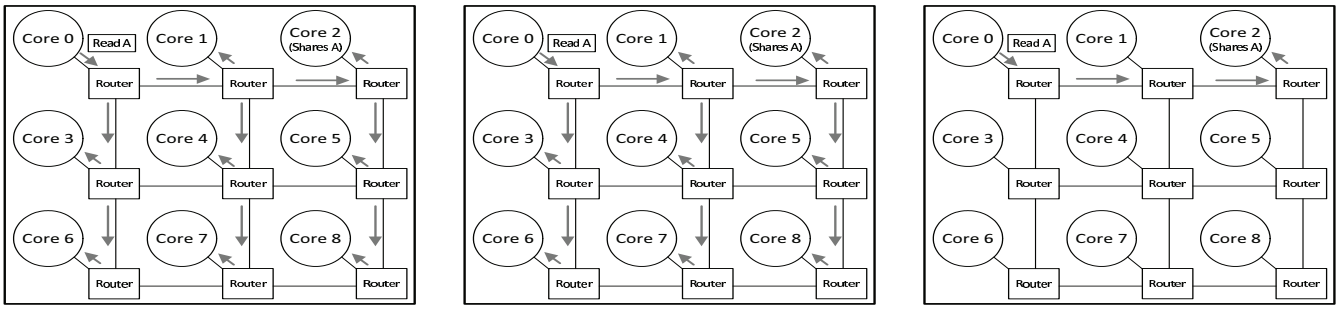
Snoopy protocols do not maintain sharing information at either the source of the request or at remote nodes, like a directory. Thus, on a cache miss, the coherence request is broadcasted to all cores to be snooped. If all of the cores are caching a block that is being snooped, a broadcast would not be wasteful. However, this is rarely the case. We performed evaluations (refer to Section 5.1 for methodology and configuration) for 16-processor CMP systems and measured the amount of redundant coherence requests in the system while running 16 threads of the same application. A redundant coherence request is one that reaches a destination core that does not share the cache line being snooped, thus unnecessarily consuming resources. Figure 1 shows the percentage of coherence requests that are redundant for different parallel multi-


*MICRO'09,* December 12–16, 2009, New York, NY, USA.




(a) No Filtering (b) Source Filtering (c) In-Network Coherence Filtering

**Figure 2: Request traversal for *non-shared* data, with different filters.**




(a) No Filtering (b) Source Filtering (c) In-Network Coherence Filtering

**Figure 3: Request traversal for data *shared* across two cores, with different filters.**

threaded applications in SPLASH-2 [28] and PARSEC [6]. We found that 72% of coherence requests, on an average, are redundant and thus unnecessarily consume network bandwidth and power, while also resulting in redundant snoop-induced cache-tag look-ups. In this paper, we target this broadcast overhead of snoopy protocols, so that we can retain the 1-hop performance benefits of snoopy protocols, while ensuring that they are viable on scalable interconnects, consuming low bandwidth, storage and power.

In the past, snoop filtering solutions have been proposed to deal with the broadcast problem of snoopy protocols. Destination filters [21,23] avoid snoop-induced cache-tag look-ups, but do nothing to avoid the interconnect broadcast bandwidth and power problem. Source-based filters [8,20] avoid broadcasts for non-shared data, but resort to broadcasting even for data that might be cached by just one or two cores. Moreover, in their current form, source-based filters only work on synchronous broadcast interconnects, like buses, and not on distributed networks. To address the above problems, we propose maintaining coarse-grain sharing information at on-chip routers that filter out redundant snoop requests, as close to the requestor as possible, and save precious interconnect bandwidth and power. Our filtering technique comprises in-network coherence filters, present at each network router, that store information on non-shared data in the system. This information is dynamically observed and adapts to changing sharing patterns. The network routers look up this information and filter out requests that are *en-route* to non-shared cores. Figures 2 and 3 show how In-Network Coherence Filtering (INCF) is effective in filtering away redundant request traffic for data that is non-shared, as well as data that is shared across a few cores. Source filters, in their current form, do not work on distributed packetized networks. Even when extended with our proposed *region updating* step (described in Section 3.2), so that they can function on distributed networks, they would still only be able to filter requests for non-shared data.

Our in-network filtering technique is successful in reducing the total number of snoops in the system by 41.9% (16-core) and 46.5% (64-core), thereby leading to a significant reduction in total network traffic: 25.4% (16-core) and 27.3% (64-core). As a bonus, request filtering in the network also results in the saving of redundant cache-tag look-ups at the destination cores, leading to lower snoop-induced cache-tag look-up power and port contention.

The rest of the paper is organized as follows. Section 2 provides background on how snoop filtering solutions work and motivates the need for tracking of coarse-grain sharing information instead of per cache line information. Section 3 describes our in-network coherence filtering proposal. Section 4 discusses the implementation details of our technique and presents the hardware design of our implementation. Section 5 discusses the evaluation methodology and presents quantitative results. Section 6 delves into prior related work and Section 7 concludes.

# 2. BACKGROUND AND RELATED WORK ON SNOOP FILTERING

In this section, we provide relevant background on previous snoop filtering proposals and motivate the tracking of coarse-grain sharing information instead of per cache line information.

## 2.1 Source vs. destination filtering

Previous works, which target the filtering of coherence requests of snoopy protocols using hardware techniques, fall under two broad categories: destination filtering and source filtering. Destination filters [21, 23] maintain filtering information at destination cores, and use this information to filter out redundant snoop requests that are going to miss in the cache and thus save on snoop-induced cache-tag look-ups. This saves cache-tag look-up power as well as reduces contention for cache-tag look-up ports. Source filters [8,20] improve upon the destination filters by also saving on interconnect bandwidth and power. These filters maintain coarse-grain sharing information at source cores[1] and eliminate unnecessary broad-

[1]Here, by source cores, we mean the cores that are requesting the cache line.

casts and snoops for non-shared data. The source-based filtering techniques work as follows. The first block access to a particular region, where region is defined as a contiguous set of addresses, is broadcasted to all cores to be snooped. All destination cores reply with the data, if present, and report whether they are caching any block from that region. If no remote core is caching the region, the source core marks the region as non-shared in a table. Subsequent misses to the blocks in the region are directly requested from memory without broadcasting to all cores. If another core starts sharing the same region, the non-shared entry at all cores are invalidated via the broadcast request of the cache line.

Source-based filters are not sufficient for future many-core snoopy protocol proposals. This is due to two reasons. Firstly, apart from saving broadcasts for data that are non-shared, saving interconnect bandwidth and power for data that are shared, but not by all cores, is also critical. As we will show in Section 2.2, the average number of sharers of a cache line is far smaller than the total number of cores in the system. Thus, a mechanism to filter out these redundant snoop requests is required. Secondly, source filters, in their current form, only work on synchronous broadcast interconnects, like buses, and not on distributed packetized networks, whereas future many-core chips with 10's to 100's of cores will likely employ a scalable on-chip network like packet-switched meshes, tori, etc. Packetized interconnects lead to source filters not having up-to-date sharing information, leading to wrong filtering decisions, thus resulting in protocol races. In practice, snoop requests can be filtered out only using up-to-date sharing information and percolating this information instantaneously to all source nodes in non-synchronous interconnects is not possible.

This paper shows that the above-mentioned problems with source filters can be mitigated by moving the filters into the interconnection network. We believe that the on-chip network is a good candidate for maintaining up-to-date filtering information, while imposing only reasonable storage requirements. We will call these filters *in-network coherence filters (INCFs)*. We address the storage problem by maintaining per-port sharing information in the INCFs that grows proportionately to the degree of an on-chip router (which can remain constant as core count increases), as opposed to the the number of cores. The problem of maintaining up-to-date sharing information in distributed on-chip networks is tackled in our work by adding a *region update* step to every request that is the first in a region by a core. This step ensures that the filtered coherence requests are indeed redundant and do not cause protocol-level races. Our *region update* step can be applied to source filters as well, so they can use it to ensure that ordering of snoopy protocols is not violated. We discuss these details in Section 3.2.

## 2.2 Regions

To determine whether a cache block is cached by a particular core, the sharing information needs to be stored somewhere so that snoop requests can be filtered out. Conventionally, information about cache coherence is maintained on a per-block granularity. However, on-chip storage is very precious and mechanisms to reduce this storage overhead are required. Previous works [8, 12, 20, 29] have proposed tracking of coherence information at a much coarser granularity than cache lines, using *regions*. A region is a contiguous portion of memory addresses and each physical cache line maps to exactly one region. What has been observed in the past is that if a cache line is not shared among a set of cores, there is a high chance that the region to which it belongs is also not shared among the same set of cores. This property of parallel programs leads to optimizations in which maintaining region-level information suffices. To confirm these observations, we did similar studies (refer to Section 5.1 for methodology and configuration) and measured the average number of sharers present

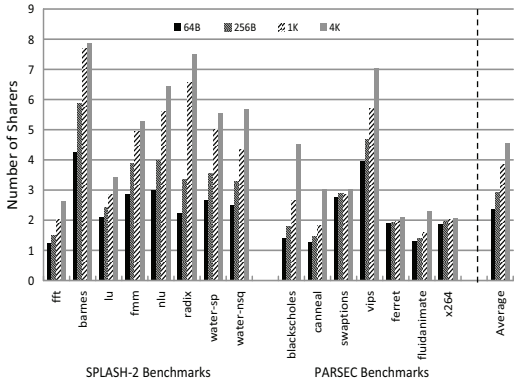


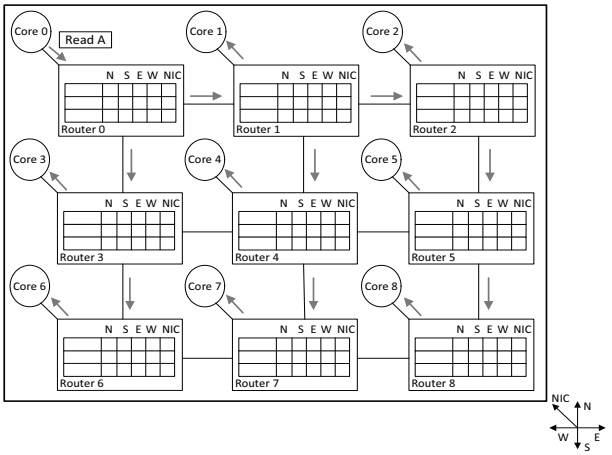

**Figure 4: Region-level sharing properties in parallel applications with varying region sizes.**

per region in parallel applications for 16-processor CMP systems. Apart from collecting block-level (where each block is 64 Bytes) sharing information, we also looked at sharing properties at a coarser granularity. The region sizes we explored were 256 B, 1 KB and 4 KB.

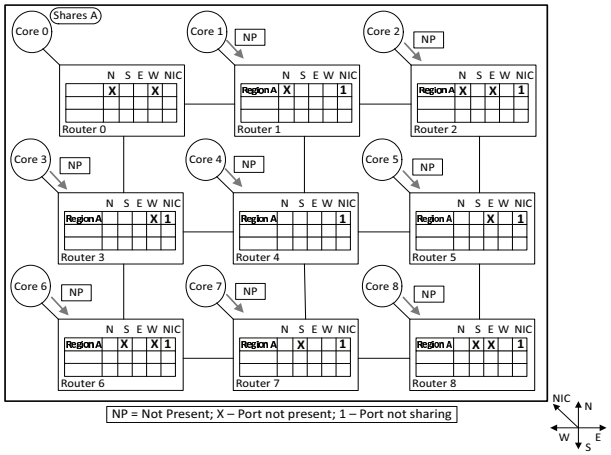
Figure 4 shows the average number of sharers per-region for various SPLASH-2 and PARSEC benchmarks. For 64 B regions, the average number of sharers was found to be about 2.4. As region size increases, the average number of sharers increases slightly. This is a result of false sharing due to coarser regions. We observe that even for 1 KB regions, the average number of sharers is about 4.0. For 16-processor systems, this means that only one-fourth of the cores share cache blocks belonging to 1 KB regions. Maintaining per-region sharing information would result in sending of snoop requests to a slightly higher number of cores, but it would result in substantial area savings, as we will show later in Section 4.1. Like previous works that used regions, we also choose 1 KB regions, for all evaluations in this paper, as a good trade-off between area savings and redundant sharing information.

# 3. IN-NETWORK COHERENCE FILTERING

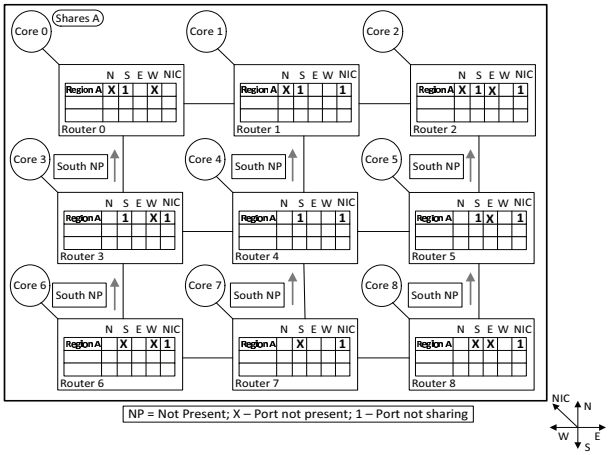
We keep INCFs at every router and maintain sharing information for each output port. Specifically, they maintain information about regions that are **not** shared by any of the cores that can be reached through an output port. This allows routers to *locally* determine which output ports are redundantly being snooped by a broadcast coherence request. The routers filter away the redundant requests, *i.e.*, they do not forward these requests onwards, thereby saving interconnect bandwidth and power. At a high level, whenever a core issues a memory broadcast request, as the request traverses the network, along the way, it informs all routers about its intention to share the cache-line. This triggers the INCFs to clear the entry corresponding to the request's region, if present. This is because an INCF entry, that is set, indicates that the region is not shared by a particular output port and this update request clears the particular bit. Since the memory request is a broadcast, all cores snoop the request and the coherence controllers present at remote cores take appropriate action. If the remote core does not currently hold any block in the corresponding region, it informs the local router to which it is attached. The local router checks its INCF to see if a non-shared entry needs to be added and takes appropriate action. It then propagates the non-sharing information to neighboring routers. Thus, over a period of time, non-sharing information spreads across the whole network and the network becomes rich with filtering information. Subsequent broadcast requests to blocks in non-shared regions will then be trimmed into unicasts, with



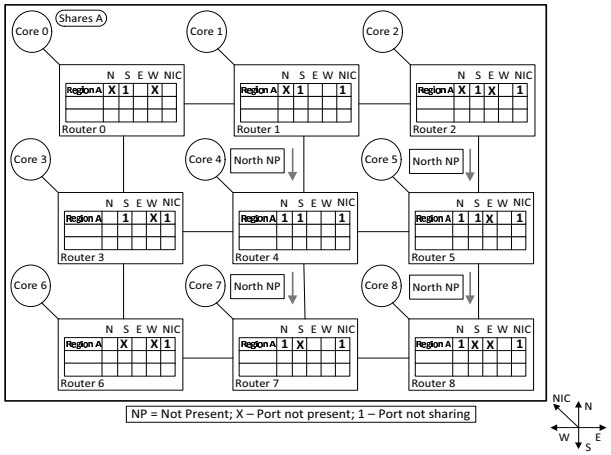

(a) Core 0 sends request Read A



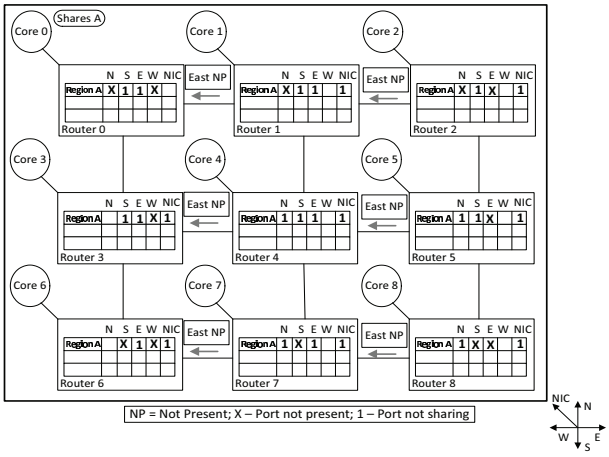

(b) Remote nodes reply that Region A is not shared



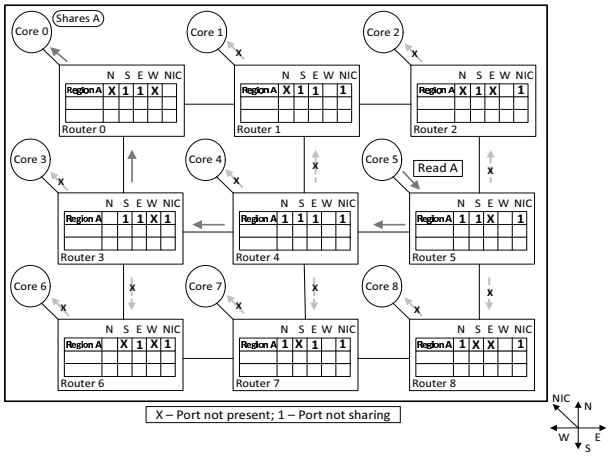

(c) Routers propagate information up North that the South port is not shared




(d) Routers propagate information down South that the North port is not shared




(e) Routers propagate information West that the East port is not shared




(f) Core 5's broadcast request filtered into a unicast to Core 0

**Figure 5: In-network coherence filtering: Walkthrough example**

each router locally filtering off redundant messages.

What allows INCFs to be effective is that only one bit of information per-port guarantees that **no** core reachable via that output port is caching the region. Moreover, the INCFs need not maintain information about all regions seen so far. We design the filters to act like a cache where older entries get evicted if there is insufficient space in the filters. Limited storage in the filters leads to some redundant requests being missed by them, but enables a practical design that is feasible within on-chip area and power constraints.

## 3.1 Walk-through example

We will next walk through our filtering technique in detail. For simplicity, we will not be tackling the ordering requirements of various snoopy coherence protocols in this example. We discuss them later in Section 3.2. Figure 5 shows the steps involved in the filtering process. We assume dimension-ordered X–Y routing for the example. In this routing protocol, all requests first traverse the East/West hops towards the destination core and then the North/South hops. An entry 1 in the INCF table indicates the corresponding port does not

share the region while an entry $X$ indicates that the particular port does not exist for the router. The entry being clear indicates that the output port is a sharer for the region. The various steps shown in the figure are as follows:

(a) Core 0 broadcasts a request, Read A, to all cores in the system to be snooped.

(b) Assuming none of the remote cores hold any cache line belonging to the region that address A belongs to, say Region A, all of them notify their local routers about their non-sharing status. The INCF tables at the local routers add an entry corresponding to Region A and set the bit at the port that leads to the local network interface (NIC), indicating that the core does not share Region A.

(c) Next, routers 6–8 notify the neighboring routers along their North direction that no core towards the South is sharing Region A. Routers 3–5 then set the bit corresponding to the South port for Region A indicating that there is no sharer for Region A towards the South. Hence, at Routers 3–5, the NIC and South bits are now set. Since the routing is X-Y, this indicates that no request coming from the North to these routers will find any shared core for Region A. Thus, they notify the neighboring routers towards North of this fact.

(d) Similar to the above step, Routers 1 and 2 forward the sharing information about the North port to routers 4 and 5. Routers 4 and 5 set the bits corresponding to the North port at the INCF table entry for Region A and forward the information southward to Routers 7 and 8. Note that Router 0 has the NIC port as a sharer and thus cannot signal the southern routers saying that nothing is shared towards the North.

(e) Routers also exchange sharing information about the East/West directions. For example, Router 5 finds that it has no sharer towards the North, South or the NIC. Thus, any message for Region A, coming from the West direction, would be redundant. Router 5 thus informs Router 4 about this. Router 4 sets the bit corresponding to the East port for Region A, indicating that there are no sharers towards the East for Region A.

(f) Now, Core 5 sends a request, Read A, to Router 5. Router 5 checks the INCF tables and filters out the North- and South-going requests and forwards the request only to the West direction. Router 4 similarly filters out the North- and South-going requests and forwards the request to the West. Router 3 finds that only the North direction needs to be snooped for Region A and forwards the request accordingly. Router 0 finds that only the NIC port is sharing Region A and thus forwards the request to Core 0. Thus, filtering leads to the request only going to shared cores and redundant message traversals being filtered off.

The walkthrough example shows how our in-network filtering technique is successful in reducing the bandwidth requirement of snoop requests to the minimum possible, i.e., just a unicast from the source to the sharer. In more general cases, INCFs filter out a significant fraction, not all, of redundant coherence requests, as will be seen from our evaluations in Section 5.

## 3.2 Ordering in various protocols

The coherence filtering technique described so far assumes that all routers have up-to-date information about sharing of regions. In CMPs with distributed on-chip networks, the information about non-shared data maintained at the cores/routers

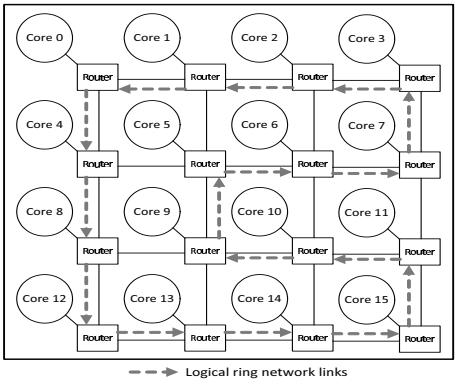



**Figure 6: Logical ring network embedded inside a physical mesh topology.**

is not updated immediately. Usage of stale information might lead to filtering of wrong messages and the cache coherence protocol not functioning correctly. In snoopy protocols, before broadcasting a request into the network, the protocol state of the cache line goes into an intermediate state and the core (say Core A) needs to see **all** subsequent requests to the cache line. In practice, remote requests that are ordered before the core's own request can be ignored. This is because the requesting core does not start owning/sharing the cache line until it sees its own request in a global order. What this implies is that all remote requests that are ordered before Core A's request can be safely filtered out. We make use of this property to ensure maintenance of up-to-date information in all our INCFs.

On a request miss to a new region, implying that the region is about to be shared and the core was not caching any block belonging to this region thus far, the core sends out a *region update* message to its locally attached router, while stalling the new request. The router updates its INCF to invalidate the entry belonging to the region, if present. This *region update* message is then sent in a snake-broadcast (like a unicast message that travels in a ring to all nodes and finally to the source node) fashion to all routers in the system. Figure 6 shows how a logical ring network is embedded in a physical mesh network and can be used by the *region update* request, thus requiring no additional physical links. The *region update* messages traverse the logical ring to visit all nodes in the system and finally back to the requestor. The remote routers also invalidate their INCF entries, while appending necessary ordering information to the *region update* message. When the *region update* message comes back, Core A releases the new request and uses the ordering information to order the current request behind every request that could have assumed that Core A is not a sharer. This extra step in the coherence protocol adds a little delay to coherence requests, but since a region switches very infrequently between sharing and non-sharing status, the overhead of this step is negligible. Our *region update* step can also be used by source filters to work on distributed on-chip networks. The ordering techniques that we propose next are effective for both source filters as well as INCF. We will describe how ordering is done for various snoopy protocols next.

**Token coherence and Intel QPI**. Token Coherence (TokenB) [17] and Intel QPI [15] are two snoopy protocols that are resilient to ordering races in the network, since they have a fall-back mechanism in case of an ordering race. Thus, our in-network filtering mechanism needs no ordering adjustment for these protocols, thereby not requiring the *region update* step. However, whenever a core starts sharing a region, the network routers need to be informed in order to update their INCF entries. This sharing update message from the source travels similarly to the non-sharing update message that destination cores send. The message first goes to the attached local router, which updates its INCF table and then forwards

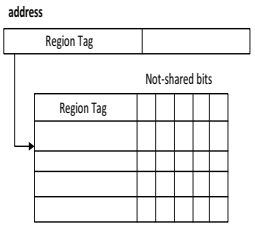



**Figure 7: INCF Table structure.**

**Pseudo code:** INCF non-share update

***Initialize:*** If no router exists in a direction, set the corresponding port bit
If NIC and South port are set:
notify North port
If NIC and North port are set:
notify South port
If NIC, North, South and East port are set:
notify West port
If NIC, North, South and West port are set:
notify East port

**Figure 8: Pseudocode for propagation of non-sharing information.**

**Pseudo code:** INCF share update

If NIC is set:
notify all ports
If South port is set:
notify North, East and West ports
If North port is set:
notify South, East and West ports
If East port is set:
notify West port
If West port is set:
notify East port

**Figure 9: Pseudocode for propagation of sharing information.**

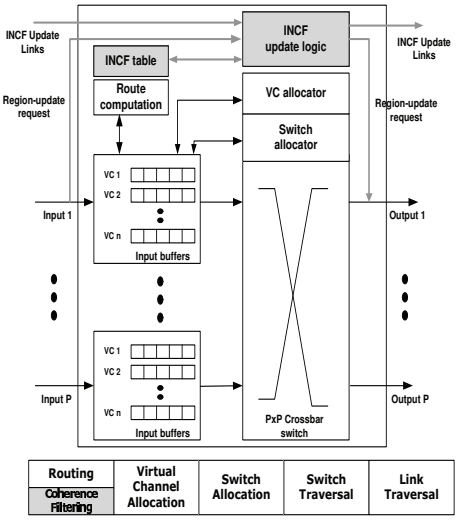



**Figure 10: Router microarchitecture and pipeline.**

the request to neighboring routers. The same network that is used by cores to inform others of the non-sharing status can be used by this update message.

**In-network snoop ordering (INSO)**. INSO [4] is a recent snoopy coherence proposal that orders snoop requests in the network. All the network routers contain pre-assigned snoop-orders (SOs) that are tagged onto requests when they arrive from the attached cores. These SOs are disjoint numbers, 0, 1, 2, etc., and the lower the SO, the earlier the ordering of a request. INSO achieves global ordering of requests by ensuring that cores process requests in SO order. As discussed above, to guarantee coherence, the request to a new region needs to be ordered behind all currently outstanding requests belonging to the same region. To ensure this for INSO, the new region request should be assigned an SO greater than the SOs of requests currently in the network that belong to the same region. More approximately, if the SO is greater than all the requests in the system right now, it should suffice. To establish this, the *region update* message is sent, which goes to the entire system and collects the lowest SO present in the routers currently. Certainly, all requests that are in-flight have an SO less than the lowest SOs in the banks. While in flight, if the request finds a higher SO, it replaces the SO it is carrying with the higher SO. When the *region update* request comes back to the requestor, it has the highest SO it has seen in the network. The buffered request is now sent to the attached router along with this tagged SO. The local attached router ensures that it assigns an SO to the request that is greater than this SO. The above steps makes sure protocol ordering is never violated due to filtering.

**Timestamp snooping**. Timestamp Snooping [18] is another snoopy coherence proposal that creates a logical ordering of snoop requests on physically unordered networks by using logical timestamps and reordering requests at the end points. NICs assign an Ordering Time (OT) to every snoop request, where OT is defined as the logical ordering time of the request. NICs also maintain a counter called Guaranteed Time (GT), which is defined as the logical time that is guaranteed to be less than the OTs of any request that may be received later by an NIC. GTs are initialized to 0 at system start-up. Timestamp Snooping provides a global order to requests by assigning OT = GT + logical time to get from the source to the furthest destination + slack, and ensuring that requests are processed by all cores in the OT order. This is achieved by ensuring NICs increment GT only after processing all requests with OT $\leq$ GT. Our objective is to order the request belonging to a new region behind all requests currently outstanding in the system. To accomplish this, whenever a new region request comes to the NIC, the NIC buffers the request and sends out a *region update* message to the network to collect the highest OT (GT + logical time to get from the source to the furthest destination + slack) in the system. Every NIC maintains this information and, hence, obtaining this information does not require any additional area overhead. OTs of all in-flight requests are guaranteed to be less than this number. When the *region update* message returns to the requestor, it gets the highest OT it has seen in the network. The NIC uses this information to assign the buffered request an OT greater than the highest OT seen in the system. These steps ensure that the global ordering of Timestamp Snooping still works with our filtering proposal.

**Uncorq**. Uncorq [24] is a snoopy coherence protocol in which snoop requests are directly broadcasted to all cores, using the shortest network path, and the global ordering among requests is done via a response message that traverses a logical ring. The logical ring is embedded inside the physical network topology. A request that reaches the supplier core, the core that owns the cache line being requested, gets ordered ahead of other competing requests to the same cache line. This is ensured by the following two properties:

1. After the supplier node processes a winning request, it does not forward the response of another request belonging to the same cache line before forwarding the response of the winning request.

2. If a node receives a positive response, the node can forward a negative response only after it has received the winning request and forwarded the winning response.

To order a new region request behind other in-flight requests, we propose the following. We will call the new requestor $R$ and the supplier core $S$. $R$ has to ensure that all active requests to the same region as the new request, whose responses have already visited $R$ in a logical ring order as well whose responses have not, get ordered before the new region request. To achieve this, $R$ broadcasts a *region update* message using the logical ring. Every node that sees this request does not snoop but simply ensures that all responses belonging to the same region as the *region update* are forwarded before it. This might require nodes to wait for the requests belonging

to pending responses. The routers attached to these nodes update their INCF tables, thereby ensuring that no newer requests would filter incorrectly. When the *region update* message returns to $R$, it is guaranteed that snoop requests, whose responses had traversed $R$ before the new region request was initiated, have been ordered ahead of the new region request. It is also guaranteed that all snoop requests, whose responses had not traversed $R$, have already done so or are pending at $R$. $R$ now frees up the new region request and converts it into a request + response message that nodes would now snoop, and gets forwarded in the logical ring. The nodes snoop the request and embed their response in the same message. They again ensure that all responses to the same region travel ahead of the new region request. When this request + response reaches $S$, it is now safe for $S$ to order it behind current responses to the same region that it has pending. Thus, every new region request requires two logical ring trips, one for the *region update* and one for the actual request + response, to complete. However, this new region request would be rare and this additional delay should impact performance negligibly.

# 4. IMPLEMENTATION

In this section, we discuss the implementation of various structures involved in our design and present the router microarchitecture required for our in-network filtering solution.

## 4.1 Region structures

To filter out redundant coherence requests, our scheme relies on the INCFs and a structure at the coherence controllers that maintains local region-level sharing information. We describe these structures next.

**INCF tables.** The central design block in our filtering scheme is the INCF that sits at every on-chip network router. As shown in Figure 7, the INCF is a simple table with entries comprising a region tag (most significant portion of the address) and $O$ bits, where $O$ is the number of output ports of the router. Each of the $O$ bits indicates whether the output port shares the region or not. While a request is going through route computation at a router, the INCF is looked up and if a matching record exists, the router knows that forwarding the request using the output port is redundant. The router avoids such forwarding and *filters out* such redundant requests. The INCF entries are evicted either as a result of limited space or when a region gets shared. Small INCF tables are sufficient for our purposes. We will show later (Section 5) that INCFs with just 64 entries are sufficient to filter away most of the redundant requests and table sizes larger than 64 lead to diminishing returns. Next, we calculate the actual overhead of the INCF tables. For a 40-bit physical address space and 1 K regions, the region tag is 30 bits long. For a conventional mesh topology, each router has five ports and, hence, each INCF table entry is 35 bits long. For 64-entry tables, the total storage overhead is only 280 B per router. To estimate the latency and power overhead of the INCF tables, we used the CACTI [25] cache model. The technology node that we assumed is 65 nm. The read access delay of a 4-way associative, 64-entry INCF table (This is what we use in our evaluations) was reported to be 0.049 ns. Thus, each table access can be done within a clock cycle for up to 2 GHz clock. The dynamic energy spent in accessing the INCF table was reported to be only 0.0025 nJ. Thus, we see that the latency and power overheads of the INCF tables are small.

**INCF updates**. The INCF table entries are updated with non-sharing information by the destination cores and are invalidated whenever a core starts sharing a region. In our current design, the source and destination cores use separate physical links to update routers about region sharing and non-sharing. As shown in the above example, each region can be encoded using 30 bits. An additional bit is required to signal whether the message is to indicate region sharing or to inform about non-sharing. Thus, the separate *INCF update links* are about 4 bytes wide. After updating its INCF table, each router's filtering logic informs neighboring routers about the sharing/non-sharing update, depending on the entries already present in the INCF table corresponding to the region. Figure 8 shows pseudo-code of how INCFs propagate non-sharing information to neighboring routers. The propagation of this information is tightly coupled with the routing protocol as it is the routing protocol that dictates which destination nodes are reachable using a particular output port. The pseudo-code assumes dimension-ordered X-Y routing. Figure 9 shows pseudo-code of how INCFs propagate sharing information to neighboring routers. This logic, like the logic to propagate non-sharing information, is used for communication between neighboring routers. These update requests are only about 4 bytes wide (region size + one bit indicating sharing) and are not like a full broadcast request. The table update logic in the routers decide whether these requests need to be forwarded to adjacent routers and this depends on the current sharing status of the ports, as indicated in the INCF tables. Our experiments show that the power overhead of these updates is only about 1%. Potentially, this router-to-router communication could be done using the data path links as well; idle cycles could be stolen from the links, in which case the extra physical links could be avoided. We will investigate this design in the future. For adaptive routing, the logic again needs to keep a record of whether **no** core reachable from an output port is a sharer. It should be noted that the other message classes can use adaptive routing, independent of the routing protocol in the request message class.

**Region tracker**. Previous works that have leveraged coarse-grain sharing information among cores have all used some kind of structure that sits alongside the last-level private cache of cores and tracks sharing at the coarse-grain level of regions. A recent proposal, Region Tracker [29], replaces a conventional tag array with region tracking structures, and achieves comparable performance to the fine-grained tag array with the same area budget. Region Tracker provides the region-level sharing information "for free." In our design, we also use a Region Tracker structure at the last-level private caches that maintain a one-bit information that indicates whether there is any block cached in that region, locally, or not. Since the caches are managed at a region granularity, a region miss in the last-level local cache implicitly indicates that a region update is necessary. Note that this information is already part of the Region Tracker structure and thus we do not add any overhead to the original proposal.

## 4.2 Router microarchitecture and pipeline

Figure 10 shows the proposed router microarchitecture and pipeline (newly added portions are highlighted in grey) that incorporate our in-network filtering design. The additions that we make to a state-of-the-art router design are the INCF table, the INCF update logic and the logic to handle *region update* requests. In a state-of-the-art on-chip router [9], the header flit of a packet arrives at an input port and goes through the *Route Computation* stage to determine the output direction. The header then arbitrates for a output virtual channel in the *Virtual Channel (VC) Allocation* stage and then proceeds to the *Switch Allocation* stage where it arbitrates for the switch. Upon successful switch allocation, the header moves onto the *Switch Traversal* stage to traverse the crossbar. It then goes through the *Link Traversal* stage to reach the next router. Subsequent body and tail flits simply follow the route and VC that is reserved by the header flit. For our design, the routing pipeline remains the same apart from the *Route Computation* stage in which the INCF table entries are checked in parallel. In case a matching entry is found, redundant messages to unshared output ports are not forwarded. In a conventional

**Table 1: Simulation parameters**

| | |
|---|---|
| Processors | 16 in-order SPARC cores |
| L1 Caches | Split I&D, 32 KB 4-way set associative, 2 cycle access time, 64-byte line |
| L2 Caches | 1 MB per core, 10 cycle access time, 64-byte line |
| Memory | 4 memory controllers, 275-cycle DRAM access + on-chip delay |
| On-chip Network | 4×4 2D Mesh, 16-byte links, 4-cycle router pipeline, |
| Orion parameters | Technology: 65 *nm*, $V_{dd}$: 1.0 V, $V_{threshold}$: 0.285 V, Frequency: 1 GHz |

router pipeline, only the header flit goes through the *Route Computation* stage, while the body and tail flits simply infer their routes by looking up the VC state table. For a packet that is to be filtered, the head flit is already in the buffer and is thus deleted once it is known that it should be filtered. The VC state table is updated to indicate that the body and tail flits corresponding to this packet have to be dropped. Thus, when the body and tail flits arrive, they are simply dropped without even writing to the buffer. Since the *region update* network is a logical ring that steals cycles from the data network links, the INCF logic in the routers is required to have output queues for the links that are part of the logical ring. We will show in Section 5 that the average load on the logical ring network is very low and thus the output queues can be very small. Deadlocks in the logical ring network are avoided using two VCs and "dateline" routing [9]. At no loads, we assume the *region update* requests spend one cycle at the routers.

## 4.3 Overhead of source filters

As we showed in Section 3.2, source-based filters can be made to work on distributed on-chip networks using our *region update* step. However, even after adding the *region update* step, source filters would be successful in filtering only requests that are not shared by any of the cores in the system. If source filters wish to save redundant requests for data that is shared, albeit by a few cores, they need to store a list of sharers for every region that is shared. The storage overhead to keep this information is proportional to the number of regions a core is caching × the number of cores in the system. Even if the source filters are designed to be conservative and have a fixed-sized table to store sharing lists for only certain regions, the list of sharers would still be proportional to the number of cores. For example, for a 64-core system, the sharing list for each table entry would require 64 × log(64) = 384 b = 48 B. If the table has 64 entries, similar to our INCFs, and the region tags are 30 bits long, the total storage overhead at each core would be 3312 B. This is an order of magnitude higher as compared to the storage overhead required for our INCFs: 280 B. Moreover, every time a core starts sharing a region, it needs to collect information about all present sharers in the system. This adds additional communication overhead. Even directories require similar storage overheads as they have to maintain a list of sharer cores. For our INCF proposal, the storage required at every router remains constant even as many-cores scale, assuming the degree of the routers remains constant. Thus, we argue that on-chip routers offer the most efficient location for storing sharing information.

# 5. EVALUATION

For all our evaluations, we perform full-system simulations using Virtutech Simics [26]. The GEMS [19] tool set is used to perform timing simulation. The Garnet [3] interconnect model is used to capture the detailed aspects of the interconnection network. The above simulation framework provides a complete cycle-level memory system timing model. We used Orion 2.0 [14] to estimate the power consumed by various components of the network. Orion models the power consumed by all the major components of the routers (buffer reads, writes, allocators and crossbar) and the network links. We also model the extra region links and the power consumed by these links.

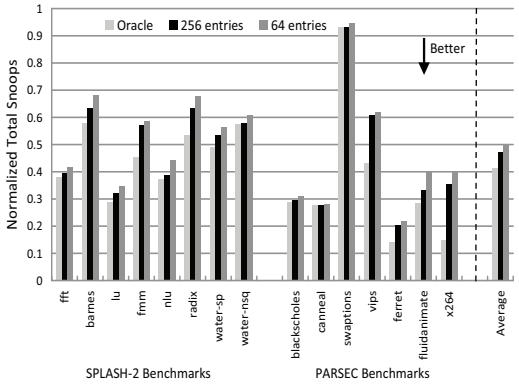



**Figure 11: Total number of snoops with various INCF filter sizes for Token Coherence.**

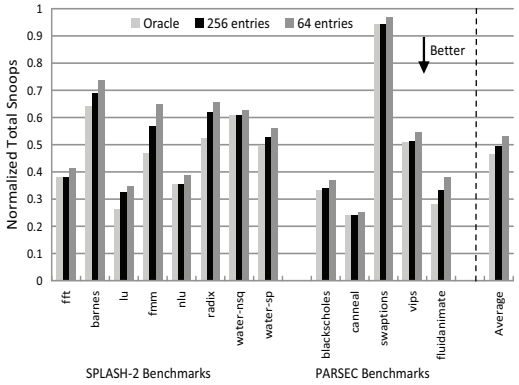



**Figure 12: Total number of snoops with various INCF filter sizes for INSO.**

We use CACTI to estimate the power consumed by the INCF table look-ups.

## 5.1 Target system and configuration

We simulated a 16-core tiled CMP architecture with the parameters shown in Table 1. Each tile consists of an in-order SPARC core with a private L1 and a private L2. DRAM is attached to the chip using four memory controllers that are at the corners of the chip. We model a 4×4 2-D Mesh network with 16-B links. The routing protocol used is dimension-ordered X–Y routing. The on-chip routers are assumed to have perfect hardware multicast support [11].

We evaluate SPLASH-2 [28] and PARSEC [6] parallel benchmarks for all the configurations. Each run consists of 16 threads of the application running on the 16-core CMP. We perform multiple runs with small random perturbations to capture the variability in parallel workloads [5]. We average the results of various runs.

## 5.2 Filtering techniques and coherence protocols

We evaluated our in-network coherence filtering technique against a design with source filtering. The source filtering design has tables, which have information about non-shared regions, at each core, and requests for those regions are directly sent to memory without broadcasting. We also did simulations

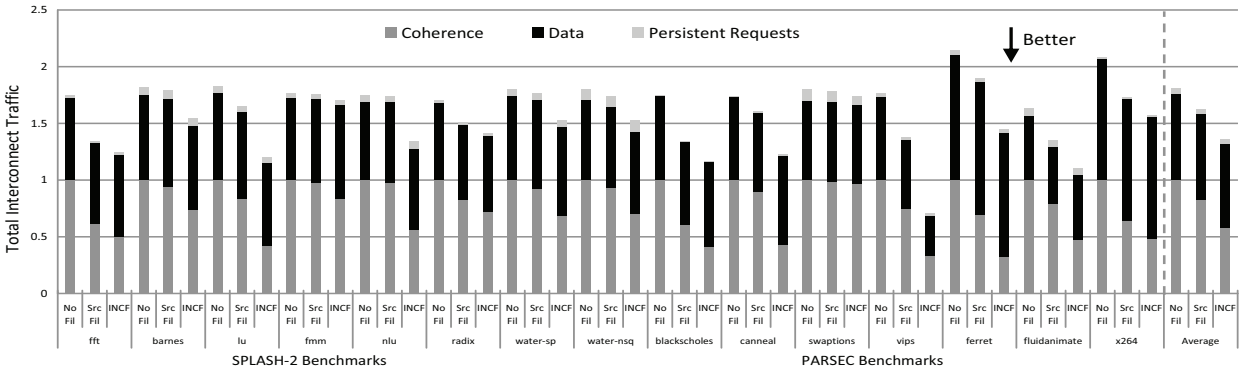



**Figure 13: Network traffic for Token Coherence.**

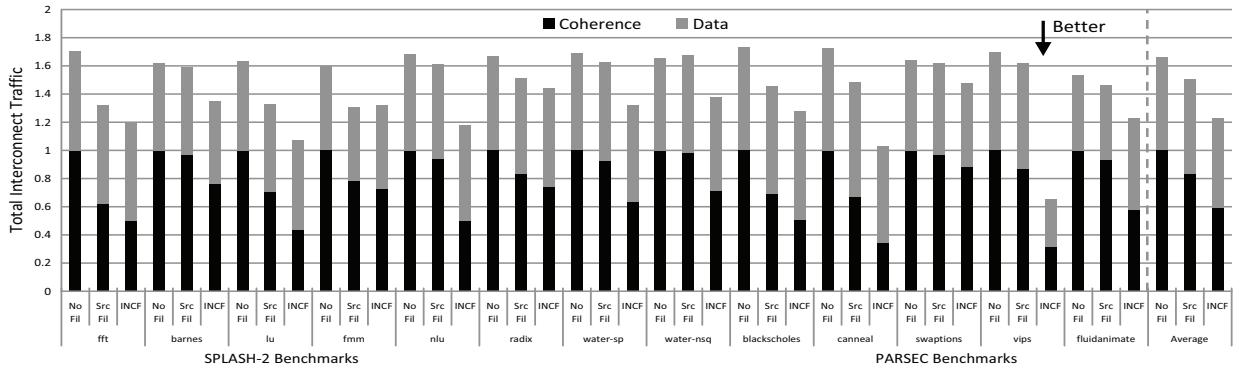



**Figure 14: Network traffic for INSO.**

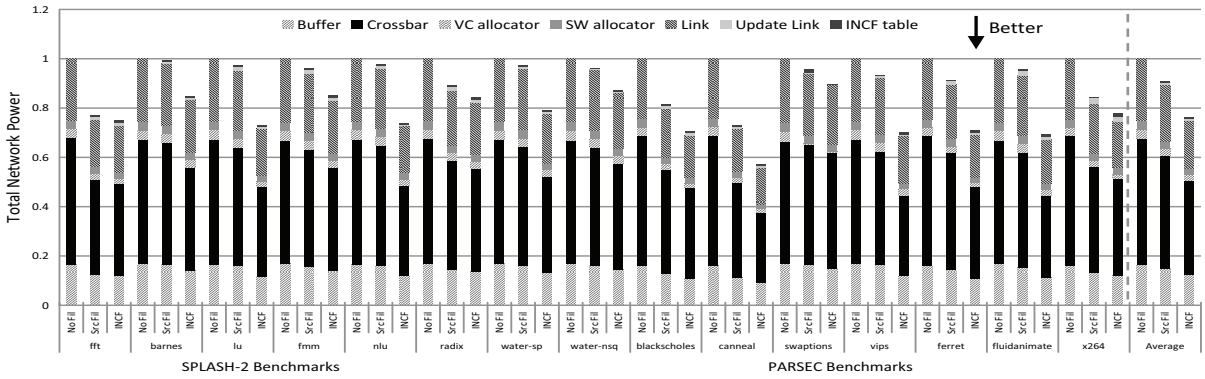



**Figure 15: Network power for Token Coherence.**

without any filtering techniques to model a baseline design. To find out the appropriate INCF table size, we explored a design with infinite table entries (Oracle) and two designs with 256 and 64 entries, respectively. When comparing the source filter with INCF, both designs had 64-entry tables at each tile (the table is at the router for INCF and at the core for source filter). All other parameters remain the same while comparing different filter designs.

We evaluated two coherence protocols on the above configurations. We chose one protocol, Token Coherence (TokenB), that does request ordering at the cache controllers and does not rely on the ordering in the network. Our *region update* step is not required for such a protocol. The other protocol, INSO, relies on the ordering of coherence requests in the network and, hence, uses our *region update* step for correct coherence ordering. The *region update* step was implemented both for the source filter and INCF. The specific coherence protocol parameters were kept constant across runs.

## 5.3 Evaluation results

We next present the evaluation results of INCF as compared to source filtering as well as a design with no filters. First, we analyze whether small INCF table sizes provide adequate filtering.

**INCF table size exploration.** Figures 11 and 12 show the normalized total snoop requests (smaller is better) in the system with INCF table sizes being infinite (Oracle), 256 and 64 entries for Token Coherence and INSO, respectively. All values are normalized to the total number of snoops without any filtering implementation. We do not plot source filters in this graph, since the aim of this exercise is to find out the appropriate table size for INCF. The figures show that with an INCF table size of 64 entries, our filtering solution is able to reduce the total number of snoops by 50% and 47%, on an average, for Token Coherence and INSO, respectively. It can also be seen that even with an infinite INCF table size, the total number of snoops can be reduced by 59% and 53.5%, on an average, for Token Coherence and INSO, respectively. This shows that INCF tables with 64 entries are reasonably close to what an Oracle INCF implementation would achieve. For the rest of the paper, we present results with INCF table size of 64 entries. Note that in one of the benchmarks, *Swaptions*,

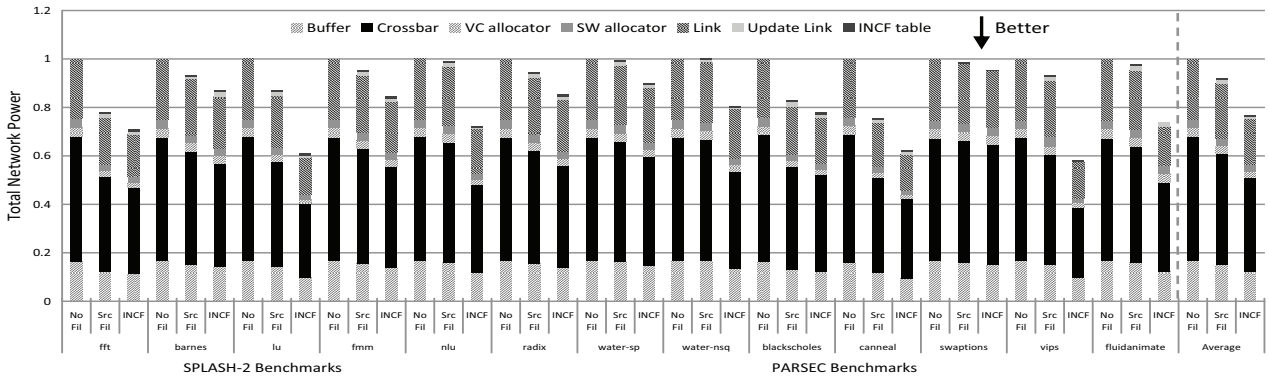



**Figure 16: Network power for INSO.**

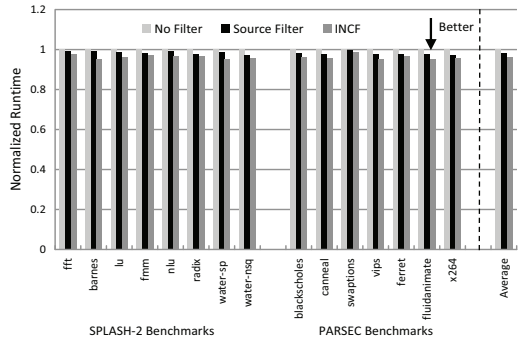



**Figure 17: Normalized runtime with and without filtering for Token Coherence.**

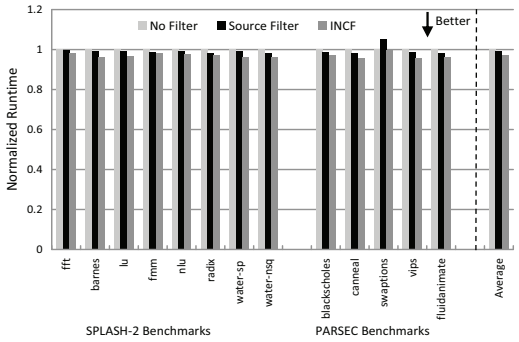



**Figure 18: Normalized runtime with and without filtering for INSO.**

for both protocols INCF has total number of snoops close to a design with no filters. This is because most of the snoop requests that enter the network are not redundant. This can be confirmed from Figure 1 that showed the total number of redundant snoop requests in the system. Although INCF tables with 64 entries are good enough for most of the benchmarks, in one particular case on Token Coherence: *x264*, INCF with 64 entries reduces the snoops by 60%, whereas an Oracle implementation would have reduced the total number of snoops by 85%. This difference in effectiveness of a smaller table size could be due to data access patterns of the benchmark having low region temporal locality. In that case, INCF table entries get frequently swapped out, necessitating a larger INCF table for the benchmark. On an average, however, INCF tables with 64 entries fare quite well.

**Reduction in network activity.** Figures 13 and 14 show the relative interconnect traffic (smaller is better) for Token Coherence and INSO, respectively, with no filtering, source filtering and INCF. For Token Coherence, source filtering reduces the coherence traffic by 17.5%, on an average, thereby reducing total network traffic by 9.9%, on an average. On the other hand, INCF reduces the coherence traffic by 42.7%, on an average, resulting in the total network traffic decreasing by 24.6%. For INSO, source filtering gets rid of 16.3% of coherence traffic, on an average, bringing down the total network traffic by 9.2%, on an average. In comparison, INCF reduces coherence traffic by 41.1%, on an average, thus reducing total network traffic by 26.2%. Thus, we see that INCF is successful in reducing the total interconnect traffic by a significant fraction. This saves network power and improves performance, as we show next.

**Reduction in network power.** Figures 15 and 16 show the relative total dynamic power consumption (smaller is better) for Token Coherence and INSO, respectively, with no

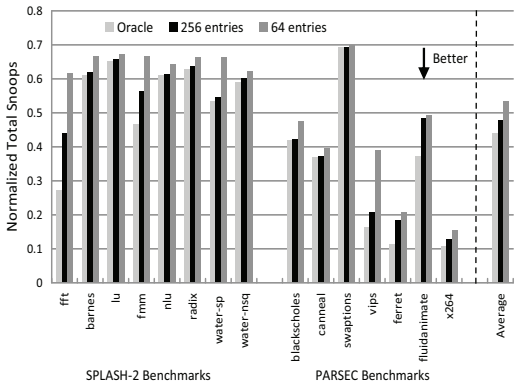



**Figure 19: Total number of snoops with various INCF filter sizes for Token Coherence on a 64-core CMP.**

filtering, source filtering and INCF. We also model the link power consumed by the links used by INCF update messages and the power consumed by region update messages. For Token Coherence, source filtering reduces total power consumption by 9%, on an average. The major power-consuming components of the network are the buffers, crossbars and links.

By filtering redundant requests, source filtering reduces the power consumption of all the three components by about 10%, on an average. In comparison, INCF reduces the total power consumption by 23.5%, on an average. It reduces the buffer, crossbar and link power by 25.2%, 25.4% and 22.6%, on an average. Both for source filtering and INCF, additional power is consumed by links used to carry INCF update messages. However, their contribution to total power is, on an average, only about 1%. This is because the INCF update messages are infrequent and only get initiated whenever a core enters

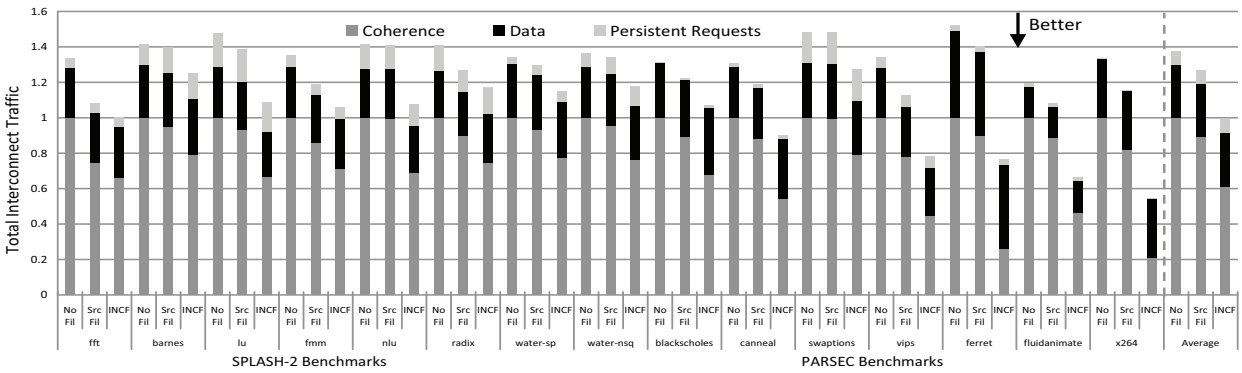



**Figure 20: Network traffic for Token Coherence on 64 cores.**

or leaves a region. Additional power consumed by the INCF table look-ups also turn out to be negligible. As highlighted in Section 4.1, INCF table lookups consume minimal energy and is thus not significant when compared to the router and link energy consumption. For INSO, source filtering is successful in reducing the total power consumption by 8.1%, on an average. On the other hand, INCF reduces the total power consumption by 23.1%, on an average. The major reduction in power is due to the reduction in buffer, crossbar and link power. Again, the additional power consumed by region links is about 1%, on an average.

**Improvement in performance.** Figures 17 and 18 show the total runtime (smaller is better) for Token Coherence and INSO, respectively, with source filtering and INCF normalized to no filtering. For Token Coherence, source filtering reduces the total runtime by 1.7%, on an average. On the other hand, INCF reduces the total runtime by 3.7%, on an average. For INSO, source filtering reduces the total runtime by 0.8%, on an average. In comparison, INCF reduces the total runtime by 2.9%, on an average. The in-order nature of the processing cores, coupled with decently sized private caches, does not stress the on-chip network. This leads to the network running mostly at low loads. Although, our filtering technique reduces the network traffic by one fourth, this does not translate into similar network performance improvements on our configurations. That is why we see limited full-system runtime improvements due to filtering. Importantly, our *region update* step does not significantly affect performance for protocols like INSO where it introduces additional delay to coherence requests to a new region. INCF leads to runtime improvement even on INSO. Interestingly, with source filtering for one of the benchmarks (*Swaptions*), the *region update* step leads to additional delay that results in total runtime with source filtering being more than without filtering by 5%. INCF for the same benchmark on INSO has similar overall runtime as no filtering. We are currently exploring evaluating our technique with out-of-order cores which would stress the network more, in which case filtering would lead to better overall performance improvement.

## 5.4 Scalability to higher core counts

To study the scalability of our filtering proposal, we performed the above experiments for Token Coherence on 64-core CMPs. All the parameters, as shown in Table 1, remain the same except the following. We simulate 8 memory controllers, which are attached to the edges of the chip. The on-chip network topology we model is an 8×8 2-D Mesh. We next present the results for table size exploration and network activity reduction.

**INCF table size exploration.** Figure 19 shows the normalized total number of snoop requests (smaller is better) in the system with INCF table sizes varied between infinite (Oracle), 256 and 64 entries. All values are normalized to the total number of snoops in the no-filtering design. With an INCF table size of 64 entries, our filtering solution reduces the total number of snoops by 46.5%, on an average. With infinite INCF tables, the total number of snoops reduces by 56%, on an average. This shows that INCF tables with 64 entries are still effective, as compared to an Oracle INCF implementation. This demonstrates that our INCF table sizes need not be increased with an increasing number of cores, in order to be an effective snoop filtering solution.

**Reduction in network activity.** Figure 20 shows the relative interconnect traffic (smaller is better) for Token Coherence, with no filtering, source filtering and INCF. Source filtering removes 10.6% of the coherence traffic, on an average, thereby reducing total network traffic by 7.6%, on an average. In comparison, INCF reduces the coherence traffic by 38.8%, on an average, resulting in the total network traffic decreasing by 27.3%. Thus, even with increasing core counts, our INCF proposal is successful in reducing the total interconnect traffic significantly, while source filtering is not able to provide significant savings. As discussed earlier, source filtering can only filter requests for non-shared data whereas INCF can filter out redundant requests even for shared data. With increasing core counts, there is a higher chance that some data would be shared by more than one core, thus rendering source filtering as an ineffective solution.

# 6. RELATED WORK

In this section, we differentiate INCF from prior proposals along three axes: filtering proposals, coherence protocols that leverage multicast, and in-network techniques that address scalable cache coherence.

**Snoop filtering.** We have already discussed various snoop filtering proposals and how INCF differs from them in Section 2. The main point is that prior snoop filtering proposals are not able to save interconnect bandwidth as well as snoop-induced cache tag look-ups for shared and non-shared data on distributed on-chip networks, while INCF is successful in filtering both shared and non-shared data.

**Multicast coherence proposals.** Multicast snooping [7] and destination set prediction [16] use prediction mechanisms to determine which processors will likely need to see a coherence request. In contrast, our work does not use any kind of prediction and simply makes use of sharing information in the network to filter out requests that are **actually** redundant. Virtual tree coherence [12] is a multicast coherence protocol that stores sharer information at virtual tree roots and all requests go to the tree root to get ordered as well as to find the sharer multicast set. In contrast, INCF works with all snoopy coherence proposals and filters redundant requests while they are *en-route* to non-shared cores. INCF does not introduce

indirection to cache-to-cache requests.

**In-network coherence.** In-network cache coherence [10] replaces conventional directories by embedding sharing information inside network routers. These tables are checked to locate data on-chip, similar to INCF. However, in-network cache coherence tries to implement the entire cache coherence substrate inside interconnect routers and runs into storage and complexity issues. In contrast, INCF provides a filtering layer to conventional snoopy coherence protocols, which are still implemented at cache controllers, and thus can be lightweight and simple. For instance, in-network cache coherence proposes a virtual tree cache of 4K entries, each entry of 27 bits, a total of about 13 KB per router, while our INCF filters take up only 280 B per router. Previous work [22] that is related to in-network cache coherence is different from INCF in a similar manner.

# 7. CONCLUSION

We observed that a major fraction of coherence requests in snoopy protocols are redundant and waste interconnect bandwidth and power. Redundant snoops also waste snoop-induced cache tag look-up power and increase contention at the destination cache tag ports. First, we proposed a region updating step that makes it possible for source filters to work on distributed networks. Second, we proposed INCF, a set of small and effective filters that sit at on-chip routers and can detect a significant percentage of redundant snoops. These redundant snoop requests are filtered to save interconnect bandwidth and power. It works with all snoopy coherence protocols and uses a *region update* step that ensures that the ordering of the cache coherence protocol remains intact and is not affected by our filtering proposal. It is fundamentally different from previous snoop filters as well as coherence protocols based on directories and multicasting. It is a filtering layer added on top of the coherence protocol to dynamically detect sharing patterns and use the information to filter redundant requests. It tackles the interconnect bandwidth problem of snoopy coherence protocols and makes them a scalable option for future many-core chips.

# 8. ACKNOWLEDGMENTS

This work was supported in part by NSF (grant no. CNS-0613074), MARCO Gigascale Research Center and SRC (contract no. 2008-HJ-1793).

# 9. REFERENCES

[1] IBM Power6. `http://www-128.ibm.com/developerworks/power/library/pa-expert1.html`.
[2] Sun Niagara. `http://www.sun.com/processors/throughput/`.
[3] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. GARNET: A detailed on-chip network model inside a full-system simulator. In *Proceedings of International Symposium on Performance Analysis of Systems and Software*, Apr. 2009.
[4] N. Agarwal, L.-S. Peh, and N. K. Jha. In-network snoop ordering (INSO): Snoopy coherence on unordered interconnects. In *Proceedings of International Symposium on High Performance Computer Architecture*, Feb. 2009.
[5] A. R. Alameldeen and D. A. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8–17, 2006.
[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2008.
[7] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood. Multicast snooping: A new coherence method using a multicast address network. In *Proceedings of International Symposium on Computer Architecture*, May 1999.
[8] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *Proceedings of International Symposium on Computer Architecture*, Jun. 2005.
[9] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Pub., 2003.
[10] N. Eisley, L.-S. Peh, and L. Shang. In-network cache coherence. In *Proceedings of International Symposium on Microarchitecture*, Dec. 2006.
[11] N. Enright Jerger, L.-S. Peh, and M. Lipasti. Virtual circuit tree multicasting: A case for on-chip hardware multicast support. In *Proceedings of International Symposium on Computer Architecture*, Jun. 2008.
[12] N. D. Enright Jerger, L.-S. Peh, and M. H. Lipasti. Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence. In *Proceedings of International Symposium on Microarchitecture*, Nov. 2008.
[13] Intel. From a few cores to many: A tera-scale computing research overview. `http://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf`.
[14] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi. Orion 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. *In Proceedings of Design Automation and Test in Europe Conf.*, Feb. 2009.
[15] D. Kanter. The common system interface: Intel's future interconnect. `http://www.realworldtech.com/page.cfm?ArticleID=RWT082807020032`, 2007.
[16] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *Proceedings of International Symposium on Computer Architecture*, Jun. 2003.
[17] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *Proceedings of International Symposium on Computer Architecture*, Jun. 2003.
[18] M. M. K. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood. Timestamp snooping: An approach for extending SMPs. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.
[19] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.
[20] A. Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. In *Proceedings of International Symposium on Computer Architecture*, Jun. 2005.
[21] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. Jetty: Filtering snoops for reduced energy consumption in SMP servers. In *Proceedings of International Symposium on High Performance Computer Architecture*, Jan. 2001.
[22] A. K. Nanda and L. N. Bhuyan. Design and analysis of cache coherent multistage interconnection networks. *IEEE Trans. Comput.*, 42(4):458–470, 1993.
[23] V. Salapura, M. Blumrich, and A. Gara. Design and implementation of the Blue Gene/P snoop filter. In *Proceedings of International Symposium on High Performance Computer Architecture*, Feb. 2007.
[24] K. Strauss, X. Shen, and J. Torellas. Uncorq: Unconstrained snoop request delivery in embedded-ring multiprocessors. In *Proceedings of International Symposium on Microarchitecture*, Dec. 2007.
[25] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. Cacti 4.0. Technical report, Hewlett Packard, 2006.
[26] Virtutech AB. Simics full system simulator. `http://www.virtutech.com/`.
[27] D. Wentzlaff, P. Griffin, H. Hoffman, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown III, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, pages 15–31, 2007.
[28] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of International Symposium on Computer Architecture*, Jun. 1995.
[29] J. Zebchuk, E. Safi, and A. Moshovos. A framework for coarse-grain optimizations in the on-chip memory hierarchy. In *Proceedings of International Symposium on Microarchitecture*, Dec. 2007.