# In-Network Cache Coherence

Noel Eisley
*Dept. of EE*
*Princeton University*
*NJ 08544, USA*
*eisley@princeton.edu*

Li-Shiuan Peh
*Dept. of EE*
*Princeton University*
*NJ 08544, USA*
*peh@princeton.edu*

Li Shang
*Dept. of ECE*
*Queen's University*
*ON K7L 3N6, Canada*
*li.shang@queensu.ca*

## Abstract

*With the trend towards increasing number of processor cores in future chip architectures, scalable directory-based protocols for maintaining cache coherence will be needed. However, directory-based protocols face well-known problems in delay and scalability. Most current protocol optimizations targeting these problems maintain a firm abstraction of the interconnection network fabric as a communication medium: protocol optimizations consist of end-to-end messages between requestor, directory and sharer nodes, while network optimizations separately target lowering communication latency for coherence messages. In this paper, we propose an implementation of the cache coherence protocol within the network, embedding directories within each router node that manage and steer requests towards nearby data copies, enabling in-transit optimization of memory access delay. Simulation results across a range of SPLASH-2 benchmarks demonstrate significant performance improvement and good system scalability, with up to 44.5% and 56% savings in average memory access latency for 16 and 64-node systems, respectively, when compared against the baseline directory cache coherence protocol. Detailed microarchitecture and implementation characterization affirms the low area and delay impact of in-network coherence.*

## 1. Introduction

With Moore's law furnishing chip designers with billions of transistors, architects are increasingly moving towards multi-core architectures as an effective way of dealing with escalating design complexity and power constraints. Commercial designs with moderate numbers of cores have been announced [1–3] with shared memory architectures maintained with snoopy cache coherence protocols. In future generations, as the number of cores scales beyond tens, more scalable directory-based coherence protocols will be needed. However, there are well-known problems with the overhead of directory-based protocols: each access needs to first go to the directory node to discover where data is currently cached, or to uncover the sharers so they can be invalidated. These traversals to and from the directory node become increasingly costly as technology scales [4]. The storage overhead of directory-based protocols is also a concern, with full-map directories taking up substantial overhead in area-constrained multi-core chips, while limited directories trade off storage with increased communication delays and bandwidth needs.

There have been a plethora of protocol optimizations proposed to alleviate the overheads of directory-based protocols (see Section 4). Specifically, there has been prior work exploring network optimizations for cache coherence protocols. However, to date, most of these protocols maintain a firm abstraction of the interconnection network fabric as a communication medium – the protocol consists of a series of end-to-end messages between requestor nodes, directory nodes and sharer nodes. In this paper, we investigate removing this conventional abstraction of the network as solely a communication medium. Specifically, we propose an implementation of the coherence protocol and directories *within* the network at each router node. This opens up the possibility of optimizing a protocol with in-transit actions.

Here, we explain in-network coherence in the light of the classic MSI (Modified, Shared, Invalid) directory-based protocol [5] as an illustration of how implementing the protocol within the network permits in-transit optimizations that were not otherwise possible

Figure 1 illustrates how reads and writes can be optimized with an in-network implementation. In Figure 1(a), node B issues a read request to the home directory node H, which then proceeds to instruct a current sharer, node A, to forward its data to node B. It consists of three end-to-end messages: B to H, H to A, and A to B. Moving this protocol into the network allows node B to "bump" into node A while in-transit to the home node H and obtain the data directly from A, reducing the communication to just a single round-trip between B and A. To investigate the potential of our approach, we characterize the ideal hop count for each read request given oracle knowledge of where the closest valid cached copy is. For this investigation, we used the 16-node simulation infrastructure of the baseline MSI directory protocol which we describe in Section 3 with the nominal configuration specified in Table 2. The baseline hop count for reads is defined as the distance from the node injecting the read request, to the home node, to the first sharer (if any) and back to the requesting node. The ideal hop count for reads is defined as the distance from the node injecting the read request to the closest node which shares the data (at the time of the read request issue), and back. If there are no active sharers, then the ideal hop count for reads is equal to that of the baseline. Results show a significant reduction in hop count of up to 35.8% (19.7% on average) can be realized for reads.

Next, Figure 1(b) illustrates optimization scenarios for write accesses with in-network coherence. In the original MSI protocol, a write request message needs to go from C to H, followed by invalidations from H to A and B and corresponding acknowledgments from A and B to H, before the request can be granted to node C. An in-network imple-

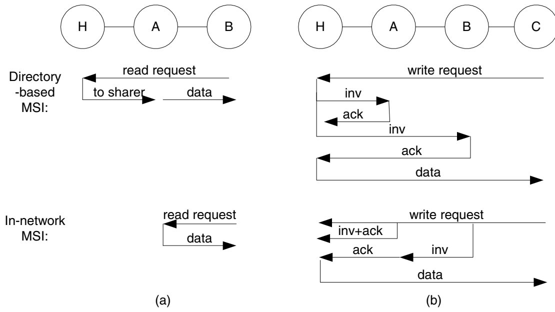



Figure 1. In-network optimization scenarios.

mentation allows A and B to start percolating invalidations and the accompanying acknowledgments once C "bumps" into them enroute to H. This *in-transit optimization* can reduce write communication from two round-trips to a single round-trip from C to H and back. To estimate the corresponding ideal hop count for writes, we assume that the invalidation of the sharer furthest from the directory node occurs when the write request is injected into the network. Thus, if the furthest sharer is farther from the directory node than the requesting node, the write request must wait at the directory node until the acknowledgement arrives before the response can be sent, and the ideal write hop count will be that from the furthest sharer to the directory node, then from the directory node to the requesting node. Otherwise, the ideal hop count for writes will be the round-trip distance between the requesting node and the directory node. Compared to the baseline of two round trips, one between the requesting node and the directory node and one between the furthest sharer and the directory node, our in-network optimization can potentially reduce up to 32.4% (and 17.3% on average) in hop count for writes.

Our contributions in this work are summarized as follows:

- We propose an approach to cache coherence for chip multiprocessors where the coherence protocol and directories are all embedded within network routers.
- Our approach has a low hardware overhead which quickly leads to hardware savings, compared to the standard directory protocol, as the number of cores per chip increases.
- Our protocol demonstrates good, scalable performace, with 27.2% and 41.2% decreases in read and write latency on average for a 4-by-4 network, and 39.5% and 42.8% improvements for reads and writes respectively for an 8-by-8 network for a range of SPLASH-2 benchmarks. Note that the improvement in memory access latency exceeds the ideal hop count reduction as it incorporates the savings garnered from not traversing extra router pipelines at intermediate and end nodes that is only possible as a result of migrating coherence into the network.

In the rest of the paper, we will describe in-network cache coherence in detail, walking through the various protocol actions, the state diagram and pseudo-code, the implementation of the router microarchitecture and pipeline, as well as how we verify its sequential consistency formally using Mur$\phi$ [6] and runtime verification in Section 2. In Section 3 we present our simulation results. Section 4 discusses and contrasts against prior related work while Section 5 concludes the paper.

## 2. In-network cache coherence

The central thesis of our in-network cache coherence is the moving of coherence directories from the nodes into the network fabric. In this work virtual trees, one for each cache line, are maintained within the network in place of coherence directories to keep track of sharers. The virtual tree consists of one *root* node ($R_1$ in Figure 2) which is the node that *first* loads a cache line from off-chip memory, all nodes that are currently sharing this line, as well as the intermediate nodes between the root and the sharers thus maintaining the connectivity of the tree. In addition, the virtual tree is always connected to the home node ($H$ in Figure 2) which is statically assigned for each memory address. The nodes of the tree are connected by virtual links (shown as $\rightarrow$ in Figure 2) with each link between two nodes always pointing towards the *root* node. These virtual trees are stored in virtual tree caches at each router within the network. As reads and writes are routed towards the home node, if they encounter a virtual tree in-transit, the virtual tree takes over as the routing function and steers read requests and write invalidates appropriately towards the sharers instead.

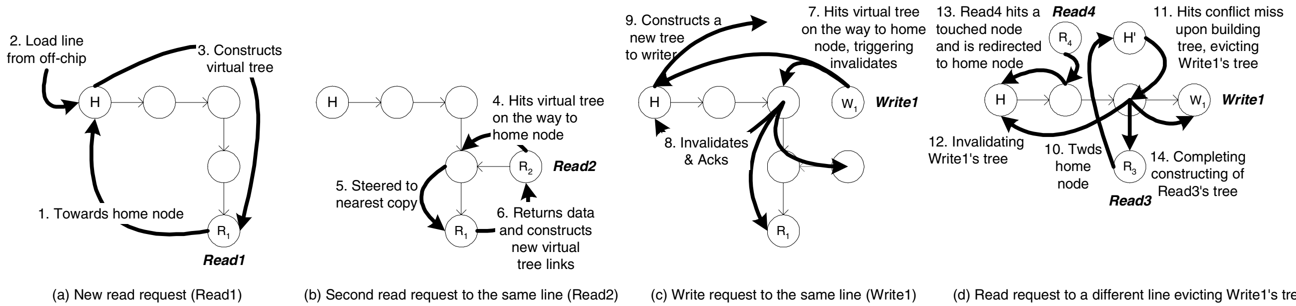
Here, as an illustration, we will discuss the in-network implementation of the MSI protocol [5], a classic directory-based cache coherence protocol. In MSI, each cache line is either *Invalid*, *i.e.* the local cache does not have a valid copy of this data; *Modified*, *i.e.* the local cache has the only copy of the cached data in the system and it is dirty, or *Shared*, *i.e.* the local cache contains a valid, read-only copy of the data, and furthermore other caches may also have read-only copies. The data cache line states remain unchanged in an in-network implementation.

### 2.1. In-network protocol

The in-network MSI protocol is illustrated in Figure 2, which depicts various scenarios that we will use to describe it. Section 2.2 follows with the detailed pseudo-code and state diagram of in-network cache coherence.

**Read accesses.** When a node reads a cache line that is not locally cached, it sends a read request message, say *Read1* of Figure 2(a). If the requesting node is not part of the virtual tree associated with the read address, the read request message is simply routed towards the home node for its address, just as is done in the baseline MSI protocol (Step 1). At the home node, one of two things may happen. First, if the home node does not have any outgoing virtual tree links, then no cache in the network contains a valid copy of the data, and therefore it must be loaded from off-chip memory (Step 2). Once the line is loaded, the home node generates a read reply message back to the original requesting node, which *constructs* virtual links towards the requesting node, pointing in the direction of the root node. The message is also routed along the physical link corresponding to the newly created virtual link (Step 3).

In another scenario, say *Read2* of Figure 2(b), should the read request message encounter a node that is a part of the virtual tree enroute (Step 4), it starts following the virtual links towards the root node instead (Step 5). Each router directs the message along the local corresponding physical link towards the root. A read request message terminates when it encounters a node, not necessarily the root, that has valid data cached for the address contained in the message, or when it reaches the home node and no tree exists, in which case the request must retrieve the data from main memory. A read reply message is then generated and




(a) New read request (Read1) (b) Second read request to the same line (Read2) (c) Write request to the same line (Write1) (d) Read request to a different line evicting Write1's tree

Figure 2. Various scenarios of reads and writes as dictated by the in-network MSI implementation. The straight arrows between nodes reflect the virtual tree links, while bold, curved arrows reflect actions prompted by the protocol.

sent back towards the original requester node, along with the data (Step 6). This *in-transit* optimization of read latency is the result of our proposed in-network implementation. When a read reply message routes back to the original requesting node, it makes the following decision at each hop along the way: if there exists a virtual link which connects to a node that is one hop closer to the requester node, it routes along the corresponding physical link. If, however, there is no such virtual link, then the message *constructs* a virtual link towards the requesting node, pointing in the direction of the root node.

**Write accesses.** A write begins similarly to a read in that a write request message is sent towards the home node (Step 7 for *Write1* message of Figure 2(c)). As in the original MSI protocol, the home node arbitrates between multiple write requests and ensures coherence by delaying the updating of a cache line until all invalidations have been sent and acknowledged.

In the simplest case, there are no virtual links for this address at the home node, and the home node does not hold a valid copy of the cache line, *i.e.*, no virtual tree exists. In this case, the home node sends a write reply message back to the original requesting node, granting permission to modify the data. Similarly to a read reply message, a write reply message constructs virtual links as it routes towards the requesting node that now becomes the root node. Once the write reply reaches the original requesting node, the local data cache line is written and the dirty bit is set. As in the original MSI protocol, if a read request or teardown message comes by later and the dirty bit is set, the cache line is written back to memory.

If, however, as in the case for *Write1*, a virtual tree exists upon arrival of the write request, then the tree must first be torn down (invalidated). Here, the in-network implementation again enables in-transit optimization of invalidation delay. Enroute to the home node, the first virtual tree node that the write request message encounters will result in the spawning of teardown messages along all virtual links off that node (Step 8). These teardown messages recursively propagate through the tree until they reach the leaves of the tree; a teardown message determines it has reached a leaf when the current node has only one virtual link. At a leaf, a teardown message is turned into an acknowledgment message and sent back up the tree towards the home node. Each time an acknowledgment message arrives at a node it removes the link on which it arrived and then the node tests to see how many virtual links remain connected to it. If only one remains then the node has become a leaf and the acknowledgement is forwarded out that link. The exception to this is the home node; all acknowledgements terminate at the home node. In this way, when the home node has no more virtual links, the entire virtual tree has been successfully torn down, and it is safe to send out a write reply message to the original requesting node (Step 9). From this point everything proceeds as discussed above for the case where no tree was found to exist for the given address. Messages which reach the home node and hit trees which are in the process of being torn down are queued until the given tree is completely torn down.

**Evictions.** In the event of a conflict miss in the virtual tree cache, the network needs to evict the entire virtual tree of which the victim tree cache line is a member. This is done to ensure that trees do not become disjoint. An eviction begins at the router at which the conflict occurs and generates a teardown message that percolates through the tree just as described above. Once the local tree cache entry has been invalidated, the original action may proceed.

For instance, say a new read request *Read3* occurs following the completion of *Write1*. It first heads towards its home node $H'$ (Step 10), and upon its return, attempts to construct a virtual tree back. Enroute, it causes a conflict miss with *Write1's* tree (Step 11), and forces its eviction, which is achieved through the percolation of teardown messages through the tree (Step 12). Once the local tree cache entry of *Write1* is successfully invalidated, the read reply message will continue and complete the tree construction (Step 14).

Because a read or write reply may have to wait for a local tree cache line to become available for a new virtual tree, that is, for the evicted tree's teardown messages to propagate out to the leaves and back to the current node, a reply message may become temporarily stalled. This stalling allows the possibility of deadlock if two or more trees are simultaneously being created and they are attempting to tear down each other. We implement a timeout-based deadlock recovery scheme: if a reply message runs into another tree and initiates a teardown of that tree, but the local tree cache line is not invalidated within a timeout interval (30 cycles for all experiments in this paper), the construction of the new tree is abandoned by transforming the reply message back into a request message at the intermediate node. At the same time, a teardown message initiates the removal of the partially constructed tree. The new request message is then held at the home directory node for a random backoff interval (between 20 and 100 cycles for all experiments). Section 3.5 explores and shows that this deadlock recovery scheme has little impact on overall memory access latency.

Should a read or write request message collide with a teardown message, such as if a read request is following a virtual tree that is in the process of being torn down, the router redirects it towards its home node, where it will wait until the tree has been completely torn down before pro-

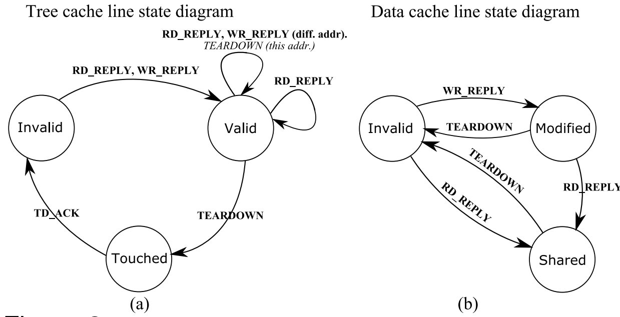



Figure 3. State diagrams of (a) the tree cache lines and (b) the data cache lines in our protocol. Bold and italicized messages denote the causes and results of transitions, respectively.

gressing again. This is shown as Step 13 in Figure 2(d). *Read4* heads towards the home node and intercepts the tree at a point where it has already been partially torn down and so it continues to the home node to wait until the tree has been completely removed.

Finally, to reduce the average time that write requests spend waiting at the home node for trees to be torn down, we implement a proactive eviction strategy. At each node a write request visits on its way to the home node, if there is no existing tree for the requested address, but the matching set of the tree cache (where a matching tree would exist) is full, it generates a teardown message for the least recently used tree in that set. This increases the chances that a write request (whether the current or a future one) will arrive at the home node and be able to begin constructing a tree without having to invalidate an existing tree.

**Victim Caching.** One of the drawbacks from having the virtual tree coherence structure distributed across multiple nodes is that a single tree can simultaneously occupy multiple tree cache lines across the network; this is in contrast to a directory-based protocol where all the information about all of the sharers exists in only one directory cache entry in the network. As a result, for equivalent directory and tree cache sizes per node, our protocol cannot actively cache as many blocks of data in the network at any given time. Therefore, we implemented an optimization where the root node of a tree sends its data to the home node when it is torn down; if the data is dirty, then it must be written back to main memory anyway, and if it is clean, then the data is piggybacked in the acknowledgement message which propagates up the tree that must eventually terminate at the home node as well. Now, whenever a read request reaches the home node, if there is no matching tree in the network, it checks the L2 cache of the home node. If the data is there, it uses that copy to send back to the requesting node; otherwise, it retrieves the data from main memory. Additionally, whenever a new tree is being constructed, the matching cache line in the local L2 cache (if it exists) is invalidated to maintain coherence (discussed further in Section 2.4). We also implemented this optimization in the baseline directory protocol in order to ensure a fair comparison.

## 2.2. Protocol pseudo-code and state diagram

In Table 1 we present a high-level version of the definition of the protocol at each node. The pseudo code describes the algorithm which operates on each message type that arrives at a router. The difference between the state diagrams of a tree cache line and a data cache line is depicted in Figure 3. The state diagram of a line in the data cache

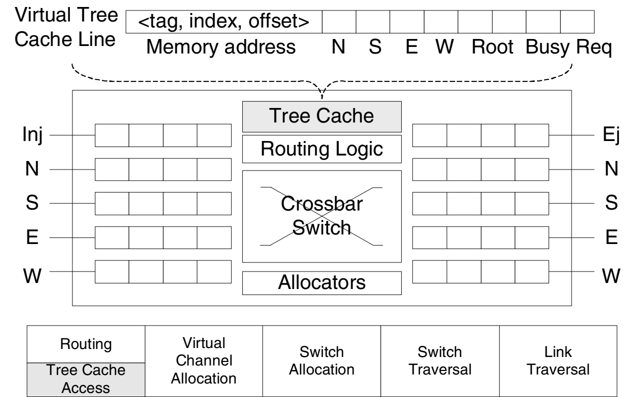



Figure 4. Proposed router microarchitecture, highlighting the structure of the tree cache line: [N,S,E,W: North, South, East, West virtual links; Root field; Busy bit; Outstanding Request bit] and proposed router pipeline, with a new pipeline stage added for accessing the tree cache.

is the same as that for the typical MSI protocol and is not altered by our in-network protocol, but transitions are now triggered by our specific in-network coherence messages. A tree cache line can be in one of only three states: invalid, valid (*i.e.* part of an active tree), or touched (part of an active tree which is in the process of being torn down). A tree cache line can only transition from an invalid to a valid state or from a touched to an invalid state by messages which have the same memory address. A tree cache line can also only transition from a valid to a touched state by a teardown message of the same address, although messages of other addresses can evict tree cache lines and generate teardown messages of the evicted address, indirectly causing the tree cache line to transition in the next cycle. Table 1 and Figure 3 show that the definition of in-network cache coherence in terms of message types permits a simple router microarchitectural implementation that we will detail next.

## 2.3. In-network implementation

**Router microarchitecture.** Figure 4 sketches the router microarchitecture and pipeline of the proposed in-network implementation. The only difference in the router microarchitecture that implements the proposed protocol versus a typical interconnection network router is the addition of the *virtual tree cache*. In a conventional network router [7], the first flit (head) of a packet first goes through the *Routing* pipeline stage to determine the output port it should request for. Next, it goes through the *Virtual Channel Allocation* stage to obtain a virtual channel for that output port. If successful, it moves onto the *Switch Allocation* stage to gain passage through the switch, before finally traversing the crossbar and link (*Switch* and *Link Traversal* stages) onto the next router. Subsequent body and tail flits simply follow the route that is reserved by the head flit, linked via the virtual channel ID stored in each flit.

In our proposed in-network implementation, a new pipeline stage is added – that for accessing the virtual tree cache. Like the *Routing* stage, it serves to steer head flits towards the appropriate output ports. Unlike in a conventional router, it does not guide head flits towards their destinations but instead points them towards caches housing the most up-to-date data requested. In the same way as a regular data cache, the memory address contained in each packet's

Table 1. Pseudo code of in-network cache coherence protocol kernel.

| **Read request (RD_REQ)** | **Write request (WR_REQ)** |
|---|---|
| **if** Current node is part of the tree and has valid data **then**<br>Get the data from the node and generate a RD_REPLY<br>**else if** Current node is part of the tree but does not have valid data<br>Route this message in the direction of the root<br>**else if** Current node is not part of the tree and is the home node<br>**if** Data is cached locally at the home node **then**<br>Get the data from the local node and generate a RD_REPLY<br>**else** Get the data from main memory and generate a RD_REPLY<br>**else** Route this message towards the home node using x-y routing | **if** Current node is the home node **then**<br>**if** A tree exists for this message's address<br>Generate a TEARDOWN message for this tree<br>**else** Generate a WR_REPLY message<br>**else** {**if** this message conflicts with an existing tree **or**<br>the matching set is fully occupied with active trees<br>Generate a TEARDOWN message for the matching tree or the LRU tree<br>Route this message towards the home node using x-y routing} |
| **Write reply (WD_REPLY)** | **Read reply (RD_REPLY)** |
| **if** Message has exceeded timeout interval **then**<br>Delete this message and Generate a RD_REQ message<br>Generate a TEARDOWN message for this tree; break<br>**if** Current node is the requester node **then**<br>**if** There is an invalid line in the matching set of the T$<br>Validate this line and set the appropriate link bit ([N,S,E,W])<br>(*T$: Invalid → Valid*)<br>Write the data to the node's cache (**D$: Invalid → Modified**)<br>**else** Issue a TEARDOWN message for the address of the LRU line in<br>the matching set of the T$<br>Wait until there is an invalid T$ line<br>**else** {**if** There is an invalid line in the matching set of the T$ **then**<br>Validate this line and set the appropriate link bit (*T$: Invalid → Valid*)<br>Route this message one hop closer to the requester using x-y routing<br>**else** Issue a TEARDOWN message for the address of the LRU line in<br>the matching set of the T$<br>Wait until there is an invalid T$ line}<br>**Acknowledgement (TD_ACK)**<br>Clear the appropriate link bit for this T$ line<br>**if** This node is now a leaf<br>Send a TD_ACK message out on the only remaining link at this<br>node for this tree<br>Invalidate this T$ line (*T$: Touched → Invalid*)<br>**else** Delete this message | **if** Message has exceeded timeout interval **then**<br>Delete this message and Generate a RD_REQ message<br>Generate a TEARDOWN message for this tree; break<br>**if** Current node is the requester node **then**<br>**if** There is an invalid line in the matching set of the T$<br>Validate this line and set the appropriate link bit ([N,S,E,W])<br>(*T$: Invalid → Valid*)<br>Write the data to the node's cache (**D$: → Shared**)<br>**else** Issue a TEARDOWN message for the address of the LRU line in<br>the matching set of the T$<br>Wait until there is an invalid T$ line<br>**else if** Current node already belongs to the tree and there is a link in the<br>tree which leads one hop closer to the requester **then**<br>Route this message out that link<br>**else if** Current node already belongs to the tree but there is no link in the<br>tree which leads one hop closer to the requester **then**<br>Route this message, using x-y routing, to the next node which is<br>one hop closer to the requester<br>Construct the new link in the tree by setting the appropriate link bit<br>**else** {**if** There is an invalid line in the matching set of the T$ **then**<br>Validate this line and set the appropriate link bit (*T$: Invalid → Valid*)<br>Route this message one hop closer to the requester using x-y routing<br>**else** Issue a TEARDOWN message for the address of the LRU line in<br>the matching set of the T$<br>Wait until there is an invalid T$ line} |
| **Teardown (TEARDOWN)** | |
| **if** There is no matching T$ line at this node **or** this T$ is *touched* **then** Delete this message<br>**else** {Touch this T$ line (*T$: Valid → Touched*); Invalidate the local data cache line (**D$: → Invalid**)<br>Generate and send TEARDOWN messages out all links in the tree except for the one on which this message arrived<br>**if** This is a leaf node **then** {Generate a TD_ACK message}; Delete this message | |

header is first parsed into $< tag, index, offset >$; if the tag matches, there is a hit in the tree cache, and its prescribed direction(s) is used as the desired output port. Otherwise, the default routing algorithm determines the desired output port. The virtual tree cache can be accessed in parallel with the routing pipeline stage and sized appropriately to lower the impact to the overall router pipeline delay while ensuring a low miss rate. Body and tail flits experience no change to the router pipeline; they still follow the route that is reserved by the head flit. Note that accessing the *data* cache, to read or write a data cache line, still occurs above-network, through the cache controllers that interface with the network. Hence, the packet will go through the router pipeline and leave through the router's ejection port, be interpreted by the interface logic which will read/write the required line and appropriate messages will be re-injected back into the router.

**Virtual tree cache organization.** As shown in Figure 4(a), each entry of the virtual tree cache consists of 9 bits (in addition to the tag bits) for a 2-dimensional network: a virtual link field with one bit per north, south, east, west (NSEW) direction (4 bits); two bits to describe which link leads to the root (2 bits); a busy bit (1), an outstanding request bit (1), and a bit designating whether the local node holds a valid copy of the data.

The virtual link bit field has a bit set for each physical link which is also a virtual link for the given address. Since a node can by definition have only one virtual link leading *to* the root node, we only need two bits to encode which link it is. The busy and outstanding request bits are both used to maintain sequential consistency and necessary even for the baseline MSI protocol. The busy bit only applies to the home node; when set, it means that the home node is *touched* and in the process of being torn down. The outstanding request bit is set if the given node has sent a request message (read or write) for that address but has yet to receive a reply.

## 2.4. Verification of coherence and sequential consistency

The baseline MSI directory protocol by itself ensures coherence but not sequential consistency. Additional constraints are needed to guarantee the latter property (see Requirement 4 below). The proposed in-network protocol can therefore be implemented with or without sequential consistency. In this work, we enforce additional requirements and ensure sequential consistency.

Here, we first discuss intuitively why in-network coherence is sequentially consistent, listing the requirements enforced in our implementation and elaborating on two scenarios which, if handled differently, could violate sequential consistency or coherence. We then discuss how we verified the sequential consistency of in-network coherence formally using Mur$\phi$ and at run-time for each simulation run.

Requirements enforced in in-network coherence to ensure sequential consistency:

1. If a request arrives at the home node and the tree is in the process of being torn down then the request message is queued until the tree

cache line has been completely removed.

2. When a read reply obtains its data from the victimized copy at the home node's local data cache, that data cache line is invalidated.
3. If there is valid data in the home node's local cache upon processing a write reply (that is, a new tree is being constructed), that data is invalidated.
4. A requesting node waits for the corresponding reply message to return before issuing the next request message.

Requirement 1 ensures that read requests do not retrieve data which is about to be overwritten (possibly reading an old value after a new value has been written to main memory) and that the construction of new virtual trees does not interfere with old trees. Requirements 2 and 3 enforce sequential consistency in the face of victim caching. Requirement 4 ensures that reads do not complete before writes of different addresses, leading to the classic violation of sequential consistency described in [8]. To further illustrate these constraints, we describe two scenarios which would violate the sequential consistency without them.

In the first scenario, consider a read request which intercepts a tree that is in the process of being torn down. This possible scenario is split into two mutually exclusive cases: either the matching tree cache line at the given node is in the *touched* state, or it is in the *valid* state. In the first case, the read request behaves as if there were no tree at all; read requests will never attempt to add links to a tree cache line in the *touched* state, or to follow a tree which it knows to be in the process of being torn down, or to transition a tree cache line from *touched* to *valid*. So the read request will proceed to the home node, where it must wait until the tree has been completely removed from the network (Requirement 1). In the second case, the read request will construct a new link from the intercepted node, or follow the links in the tree towards the root node. This is allowed because if a teardown message comes along after the read request has constructed a new link, the teardown message will simply propagate out the new link as if it had been a part of the tree from the start.

For the second scenario, consider the victim caching optimization. This is coherent because there is never a valid victim in the home node's data cache *unless there is no active tree in the network*. This follows from Requirements 2 and 3. Thus, the value of the cached victim data can never differ from the current copy in main memory. As a result, a subsequent read to this copy is equivalent to a read from main memory.

**Formal verification.** We use Mur$\phi$ [6], a model checking tool, to verify the sequential consistency of the backbone of our proposed in-network MSI implementation. We specified state variables of the protocol, the transitions permitted, as well as the rules to be checked exhaustively in the Mur$\phi$ specification language. Due to the exhaustive nature of the search, Mur$\phi$ confines its applicability to finite-state machines. We identified and verified the micro-operations of data access, *e.g.*, read/write, and corresponding node/network operations, down to the detail of the actual message types. We encoded rules including sequential consistency constraints; for example, write operations to the same memory address must be observed in the same order by all the processor nodes. In this Mur$\phi$ model, to allow for tractable analysis, we permit multiple concurrent reads and up to two concurrent writes. Our Mur$\phi$ specification consists of 1227 lines of code, and generates on the order of 100,000 states during the model-checking. The end result showed Mur$\phi$ verifying that our in-network MSI implementation is sequentially consistent.

**Runtime verification.** Finally, to give more confidence in our protocol, we log every memory operation as it occurs in our simulations to verify the coherence. As each read reply returns to its requesting node and the value is written to the local data cache, we check the value being written to the data cache against the value held in main memory. Our simulations showed no instances of a read which returns a value different from main memory.

To verify sequential consistency, at runtime we generate a total order of all tree accesses. An access is defined as occurring when a value is read from main memory or from an existing tree, or when a new tree is created to satisfy a write request. We generate a program order of all memory access replies for each node in the network. These replies are defined as occurring only when a read reply or write reply message reaches the requesting node (and the local data cache is updated). Sequential consistency is enforced if the program order (the sequence of reads and writes) of each node appears *in that order* in the total order. Our simulations showed that this condition holds true across all runs.

# 3. Simulation Results

We implemented a trace-driven architectural simulator which models the detailed on-chip memory-network microarchitecture of CMPs. Within the CMP, each on-chip processor is equipped with a two-level local cache that interfaces with a router. These on-chip routers form a coherent networking layer supporting the distributed on-chip caches. Across all addresses, the home directory nodes are statically assigned based on the least significant bits of the tag, distributed across all processors on the entire chip. We chose to implement trace rather than execution-driven simulation to allow for tractable simulation times, given the large number of experiments run. Full-system simulation is very slow and scales superlinearly with the size of the network.

The memory access traces are gathered by running a set of SPLASH-2[1] benchmarks [9] on Bochs [10], an x86 instruction-level multiprocessor simulator with embedded Linux 2.4 kernel, similar to the methodology used in [11]. Using Bochs, our baseline configuration is a 16-way CMP connected in a 4-by-4 mesh topology. Each benchmark is spawned into 16 threads and executed concurrently among the CMP.

Our simulator tracks read and write memory access latencies, which include the round-trip delay from on-chip caches through the network interface, the network (including contention), and off-chip accesses to main memory. For the directory-based MSI protocol, the latencies include the access time through the network interface to directories at the home nodes. For the in-network implementation, this includes the delay contributed by each virtual tree cache access at intermediate routers.

## 3.1. Performance of in-network cache coherence

Table 2 shows the detailed simulator configuration used in this experiment. For the directory-based protocol, each router consists of five pipeline stages. For comparison, the Alpha 21364 router has a 13-stage pipeline, 6 of which are delay cycles associated with the chip-to-chip link interface that is not applicable to on-chip networks, two are error-correcting cycles, while the router pipeline takes up five cycles [7, 12]. To support in-network cache coherence the

[1] Abbreviations used throughout the paper: bar, barnes; rad, radix; wns, water-$n^2$; wsp, water-spatial; ocn, ocean; ray, raytrace.

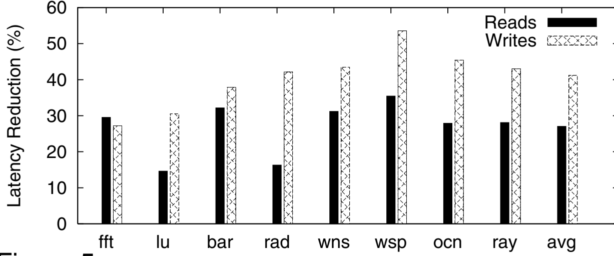



Figure 5. Reduction in average read and write access latencies for the in-network implementation of the MSI directory protocol as compared to the standard directory protocol.

virtual tree cache lookup introduces extra delay, which is a function of the tree cache configuration. The best tree cache configuration was arrived through a detailed investigation of alternative tree cache organizations (Section 3.2): a 4K-entry virtual tree cache that is 4-way set-associative. The virtual tree cache is assumed to be maximally ported; there is a read and write port for each of the 5 router ports, as having fewer ports requires additional logic for dynamically arbitrating between these ports which will hurt the critical path of the router pipeline. The tree cache size is calculated based on the number of entries as well as each entry's size, which consists of the 19-bit tag and the 9-bit virtual tree cache line. Evaluations of the tree cache access time were performed using Cacti [13] (Table 3 of Section 3.2). For this best-performing configuration, the access time is two clock cycles. As shown in Figure 4(b), the virtual tree cache is accessed concurrently with routing lookup, so for the in-network implementation, the router pipeline increases from 5 to 6 cycles. For a fair comparison, the baseline directory-based cache coherence protocol is also configured with a 4K-entry, 4-way set associative directory cache at each node's network interface.

For the memory hierarchy, we assume a 2MB L2 cache per on-chip node, and 4GB off-chip main memory. Access latency to the L2 cache is derived from Cacti to be 6 cycles, while off-chip main memory access delay is assumed to be 200 cycles. For the directory-based protocol, directory access latency is modeled as a 2-cycle round-trip (from ejection port back to injection port) through the network interface at the cache controller after the router traversal.

The average read and write memory latency reductions are presented in Figure 5. Among these 8 benchmarks, the proposed in-network cache coherence protocol reduces average read access latency by 27.1% and up to 35.5%. Write latency is reduced by 41.2% on average and up to 53.6%. As discussed in Section 1, our protocol provides a greater maximum possible savings for reads than for writes, so it is somewhat counter-intuitive that for all but one benchmark in Figure 5 the average write latency reduction is greater than that of the reads. However, this is readily explained by the fact that these benchmarks do not exhibit a great amount of data sharing. In fact, for all but one of the benchmarks, greater than 90% of the virtual trees created span only one or two valid and shared copies of a particular line of data. Thus, there are relatively few opportunities for in-transit "bumping" into shared cached data enroute to the home node. Significant savings for reads are still achieved because a reduced hop count is just one part of the overall potential for latency reduction; the other is the advantage of not having to leave the network layer as frequently. In our case, when a read request reaches the directory node, it is immediately redirected in the direction of the current copy of the data, assuming a hit in the tree cache; this scenario is compared to the baseline protocol wherein the read request must leave the router, access the directory cache on the local node, and then be reinjected into the network, whether there is a current copy of the data in the network or not. This results in an extra router pipeline traversal as well as the cost of accessing the directory cache. In our protocol writes exhibit the same opportunities for eliminating extraneous router pipeline traversals, but in addition to this, there is a significant hop count reduction due to the proactive evictions that are triggered by in-transit invalidations: as write requests travel to the home node, they initiate the tearing down of trees that are likely to lead to capacity conflicts. Thus, when a write request reaches its home node, it will spend fewer cycles on average waiting for an available tree cache line before constructing a tree.

Table 2. Simulated memory-network configuration. System is clocked at 500MHz at 0.18$\mu$m.

| Tree or directory cache config. | Value | Network-memory configuration | Value |
|---|---|---|---|
| Entries | 4K | Base router pipe. | 5 cyc. |
| Associativity | 4-way | Routing | X-Y |
| Access latency | 2 cyc. | L2 cache line size | 8 words |
| Read ports | 5 | L2 cache assoc. | 8-way |
| Write ports | 5 | L2 Access | 6 cyc. |
| Size (per entry) | 27 bits | Directory access | 2 cyc. |
| eviction policy | LRU | Main mem. access | 200 cyc. |

The other noteworthy trend of Figure 5 is the variation in average read access latency reduction. Particularly, `lu` and `rad` exhibit the least amount of savings. In our protocol, we would expect to see a correlation between the size of the virtual trees and the observed savings since if each tree spans more active sharers, it is more likely that subsequent read requests will run into such an active sharer and avoid a trip all the way to the directory. Indeed, we observe that `lu` and `rad` have the two lowest average active data copies per virtual tree (both 1.07) while the two benchmarks with the greatest read latency reduction, `bar` and `wsp`, have amongst the most average average data copies per virtual tree (1.16 and 1.33).

Write accesses, on the other hand, since they do not incur the penalty of accessing main memory, are less sensitive to the size of the trees than to the frequency with which they are torn down. This frequency is heavily dependent on the distribution of the home nodes of the accesses. That is, how many accesses' addresses map to node 0, how many to node 1, *etc.* If a particular node or nodes must handle a disproportionate number of write accesses, the write requests are more likely to queue up at the home node. Since an invalidation takes longer for the directory protocol, this penalty is paid for the current write request as well as all others that are queued at the home node, increasing the disparity in the average write access time. We measure the deviation from a perfect distribution for each of the benchmarks by calculating the root-mean-squared difference between the distribution of a given benchmark and a perfect distribution (all nodes handle $100\%/16 = 6.25\%$ of the accesses). Indeed we see that `wsp`, which saw the largest reduction in write latency, has the greatest deviation amongst the accesses, and `fft` and `lu`, which experienced the least reduction, had the least deviation from a perfectly even distribution.

### 3.2. Design exploration of virtual tree cache

Next, we explore the microarchitectural design of the virtual tree cache for the proposed in-network coherence pro-

Table 3. Access time and area overhead for tree caches of different sizes and associativities at 0.18$\mu$m, clocked at 500MHz.

| Size (no. entries) | 0.5K | 1K | 2K | 4K | 8K | 16K |
|---|---|---|---|---|---|---|
| DM: Access (cyc.) | 2 | 2 | 2 | 2 | 3 | 4 |
| 2-way: Access (cyc.) | 2 | 2 | 2 | 2 | 3 | 4 |
| 4-way: Access (cyc.) | 2 | 2 | 2 | 2 | 3 | 4 |
| 8-way: Access (cyc.) | 2 | 2 | 2 | 3 | 3 | 4 |
| 16-way: Access (cyc.) | 2 | 2 | 2 | 3 | 3 | 3 |
| DM: Area ($\mu m^2$) | .059 | .13 | .22 | .56 | .87 | 1.73 |
| 2-way: Area ($\mu m^2$) | .058 | .12 | .29 | .51 | .89 | 1.14 |
| 4-way: Area ($\mu m^2$) | .062 | .10 | .22 | .51 | .88 | 1.17 |
| 8-way: Area ($\mu m^2$) | .082 | .11 | .17 | .40 | .68 | 1.17 |
| 16-way: Area ($\mu m^2$) | .13 | .15 | .20 | .57 | .72 | 1.21 |

tocol. The design trade-offs of the virtual tree cache are mainly affected by the following parameters. The size of the virtual tree cache (number of entries) determines the capacity of the virtual tree network: more virtual trees translates to the in-network implementation supporting a greater amount of coherence sharing and fewer forced evictions due to capacity misses, thus improving performance. The associativity of the virtual tree cache affects the tree cache utilization: by increasing cache associativity, conflict misses decrease (up to a point, as we will see below), again leading to fewer unnecessary invalidations and better performance. Large, highly-associative tree caches, however, have a detrimental impact on access time and area. Since the tree cache access time is on the critical path of the router pipeline (See Section 3.8) and area is a scarce resource in chip multiprocessors, the tree cache organization needs to be judiciously traded off with the performance impact.

Table 3 shows the impact on access delay and area for various configurations of the virtual tree cache, derived from Cacti [13]. As mentioned earlier, the number of read and write ports are assumed to be the maximum (5) to obviate arbitration delay from the tree cache access time and minimize the impact on router pipeline.

Figure 6 graphs the effect of varying cache sizes on average read and write access latencies, with the associativity kept constant at 4-way, for all SPLASH-2 benchmarks tested. In order to investigate the behavior of the underlying protocol, we disable the victim caching as described in Section 2.1. As we expect, Figure 6 shows that reducing the tree cache size results in steadily increasing average read latency. The reason for this is that with smaller caches, trees are evicted from the network more frequently and so there are more read requests injected into the network which ultimately must obtain the data from main memory, incurring the large 200-cycle penalty. Since writes do not incur a penalty for off-chip access, the average write latency is not sensitive to the size of the tree caches, as we see in Figure 6. From a performance point of view, we would like to choose as large a tree cache as is practical; we keep this in mind as we continue our exploration.

Next, to uncover the optimal associativity, we vary virtual tree cache associativity while keeping capacity constant at 4K entries per node. Figure 7(a) shows that for most of the benchmarks, the average read latency decreases as we increase the associativity from 1-way (direct-mapped) to 2-way and to 4-way, but then increases as associativity is increased to 8-way. We expect the first trend because as the associativity decreases, there are more conflict misses, which result in shorter average tree lives, which in turn causes a higher percentage of reads to have to obtain data from main memory instead of hitting an existing tree. However, it is unusual that the average read latency increases when the associativity is increased from 4-way to 8-way. This can be explained by the proactive evictions triggered by our protocol. For any given existing tree in the network, if the sets are larger, then there is a greater chance that a request message traveling through the network will map to that set and possibly proactively evict that existing tree. This increases the chance that a future read, which would have hit the tree, will now miss and have to get the data from main memory. In short, a low associativity leads to more conflict misses, while a high associativity leads to more proactive eviction misses. In Figure 7(b) we see that the average write latencies of each benchmark follow the same trends as the average read latencies of the same benchmark.

The above experiments prompt us to select a virtual tree cache with 4K entries which is 4-way set associative. As seen from Table 3, such a cache can be accessed within 2 cycles, lengthening the router pipeline by just one cycle, and it has an area overhead of 0.51 $\mu m^2$. Compared to a 16-processor chip fabricated in the same 0.18$\mu m$ process, the MIT Raw chip [14], whose size is 18.2mm by 18.2mm, with 2mm by 2mm tiles, the area overhead of the virtual tree cache is negligible.

### 3.3. Effect of L2 cache size

Given that we cache victimized data at the home node's L2 data cache, we expect that if the size of the L2 cache is small, relative to the tree cache size, then we will observe decreasing performance gains as compared to the baseline directory protocol. Indeed, in Figure 8(a) we observe this trend. In fact, with a 128KB local data cache per node, our protocol performs worse than the baseline protocol for `rad` and `ray`. For these and other benchmarks which experienced greater than average decreased performance, it is due to the larger memory footprints of the applications. This is as expected because there will be less room for victimized data. Although we do see this performance loss when using a 128KB L2 cache, note that with 32-Byte lines, this is only 4K entries, or the same number as the tree caches. It is not likely that data caches will be this small in actual CMPs. Finally, as we see from Figure 8(b), the average memory access latency for writes is not sensitive to the size of the L2 caches for the same reason as before: write accesses never experience off-chip memory access penalties and must generate a new tree every time.

### 3.4. Performance scalability

Here, we evaluate the performance scalability of in-network cache coherence by scaling to a chip multiprocessor with 64 processors. We first run each benchmark, parallelized 64-way. Figure 9 shows the relative savings in average memory access latency for the in-network implementation vs. the baseline directory-based protocol for each individual benchmark across the 64 processors. We see that we are able to gain 35% savings in average read access latency and 48% savings in average write latency relative to the baseline. This shows that the percentage performance improvement continues to be maintained as network size scales up, attesting to the performance scalability of the in-transit optimizations of in-network coherence.

Most of the results in Figure 9 are comparable to those in Figure 5, with a few notable exceptions. In some cases, our protocol performs much better than in the 16-node case.

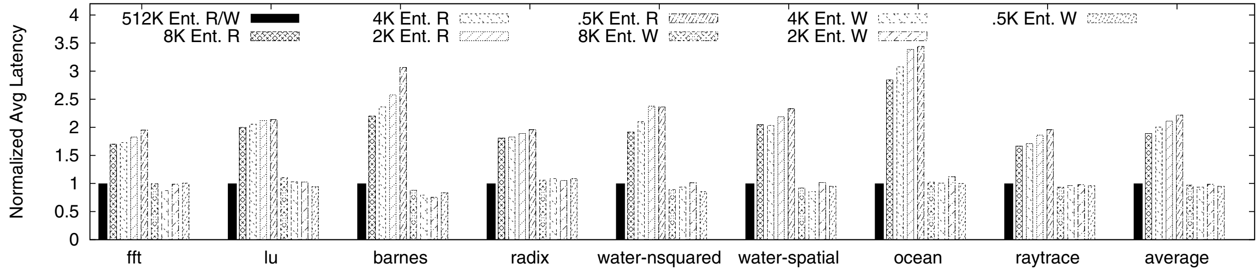



Figure 6. Effect of varying the tree cache size (number of entries) on the average latency of in-network coherence reads (top) and writes (bottom). All other parameters remain as per Table 2. Latencies are normalized to the 512K entry configuration for each benchmark.

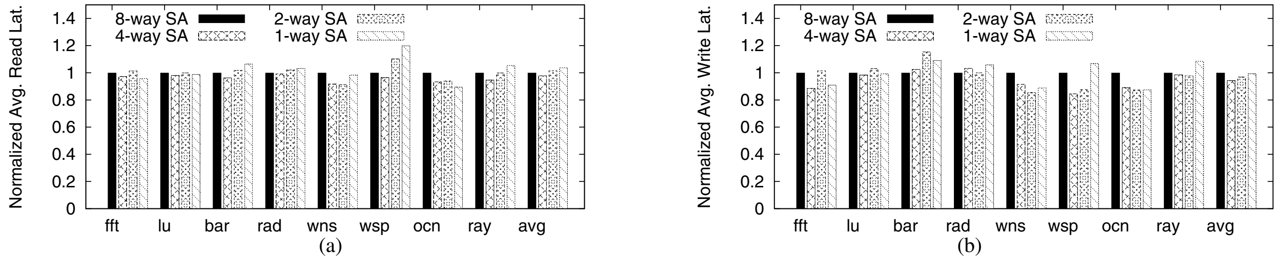



Figure 7. Effect of varying the tree cache associativity on the average latency of in-network coherence (a) reads and (b) writes. All other parameters remain as per Table 2. Latencies are normalized to the 8-way set-associative configuration for each benchmark.

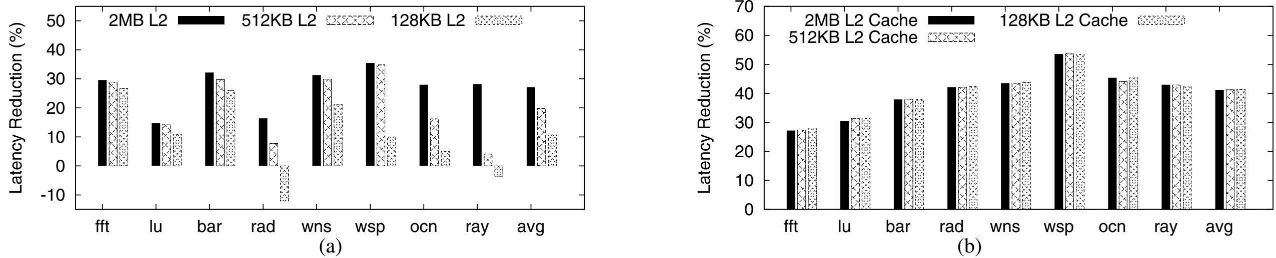



Figure 8. Effect of reducing the L2 data cache size. Percentage Latency reduction is compared to the baseline directory procotol with similarly varied cache size.

This is because the effects of contention and distance are amplified in a 64-node network; for reads, more frequent accesses lead to more conflict misses and thus more trips to main memory. For writes, more frequent accesses lead to more write requests having to queue at the home node as before, but now that the average distance between home node and sharers is larger, the cumulative queueing effect as described in Section 3.1 is much larger. Thus, it is not surprising that the two benchmarks that experienced write latency savings significantly greater for 64 nodes than for 16 nodes, `lu` and `ocn`, injected averages of .27 and .46 write requests per cycle (rpc) respectively, and that the rest of the benchmarks injected at most .16 write rpc. Similarly, `rad`, which observed 71.4% read latency savings, injected an average of 1.47 read rpc, whereas the next highest injection rate was .61 rpc.

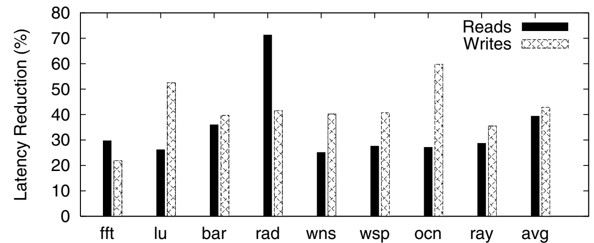



Figure 9. Average memory latency reduction for the tree protocol compared to the baseline MSI directory protocol in a 64-node mesh network.

## 3.5. Effect of deadlock recovery

In Section 2.1 we described the method of deadlock detection and recovery in our protocol. Here we quantitatively demonstrate the minimal impact that deadlock has on performace. Table 4 lists the percentages of the overall read and write latencies which are attributed to time spent detecting and recovering from deadlock (this includes the timeout intervals and the random backoff intervals). While the penalty for an individual request that deadlocks can be severe (on the order of the cost to access main memory), deadlocks occur so infrequently that it amounts to a small percentage of the overall latency. Indeed, for the nominal tree cache configuration, the only benchmarks which experienced *any* deadlock were `ray` and `ocn`. Therefore, the data in Table 4 were generated using *direct-mapped* 4K-entry tree caches, and we see that in this case, on average, deadlock accounts for just 0.20% of the overall memory access latency.

Table 4. Percentages of read and write latencies which are the result of deadlock.

| Bench | fft | lu | bar | rad | wns | wsp | ocn | ray | avg. |
|---|---|---|---|---|---|---|---|---|---|
| Rd. Lat(%) | .14 | .05 | .13 | .43 | .05 | .74 | .12 | .04 | .21 |
| Wr. Lat(%) | .15 | .14 | .19 | .03 | .06 | .61 | .08 | .05 | .20 |

### 3.6. Storage scalability

There are two key differences in our in-network implementation of the MSI protocol that affect the storage overhead and scalability. First, in our in-network implementation, the home node no longer stores a list of sharers, but only points in the direction of the root node. Now, though, each intermediate node needs storage for the virtual tree cache bits. Note however that the in-network implementation enables the storage of *directions to sharers* rather than *actual sharer addresses*, with each node only knowing about itself and its immediate neighbors. As a result, the virtual tree cache line size grows with the *dimension* of the system (the number of immediate neighbors), rather than the total number of nodes. Comparing our protocol to other common approaches, full-map and limited directories [15] require O(N) and O(logN) bits per entry. Coarse-vector schemes require O(N/K) bits, where K is the number of processors per group. Linked-list (such as SCI [16]) or tree-based protocols require O(S*logN) bits, where S is the number of sharers. In contrast, our protocol uses O(Hop(S)*logP) bits, where Hop(S) is the hop count between the furthest sharers, and P is the number of ports per router. Note that this does not depend on N.

Next, we quantify the storage overhead of our protocol as we scale the number of processors. To calculate the storage overhead of our protocol, we compare the virtual tree cache size with that of the full-map directories in the directory-based protocol. For a 16-node system, our in-network implementation uses 4K*28bits per node while the directory-based protocol uses 4K*18bits[2] per node, resulting in 56% more storage overhead for the in-network implementation; for a 64-node 8-by-8 system, the in-network implementation still uses 4K*28bits per node, but the directory protocol now uses 4K*66bits per node, so the in-network implementation now uses 58% fewer storage bits.

### 3.7. Effectiveness of in-network implementation

Throughout this paper, we advocate the effectiveness of breaking the abstraction of the network as solely a communication medium, and embedding coherence protocols and directories within the network routers. Here, we evaluate the effectiveness of such an in-network implementation versus an implementation where the virtual tree cache is housed *above* the network, at the network interface of the cache controller. This approximates the implementation of GLOW [17] (see Section 4), where packets are snooped at the network interface and reinjected and redirected for in-transit optimizations of reads.[3] To model this, we modified the simulator so each packet accessing the virtual tree cache has to first be ejected and then re-injected. Figure 10 shows how this additional delay incurred at each router hop significantly affects overall memory access latency by an average of 31% for reads and 49.1% for writes. Note that the performance impact is relatively constant across all benchmarks. This makes sense because the only difference between the two experiments for each benchmark is the effective access time of the tree cache.

[2] The busy and request bits are common to both.

[3] Note though that GLOW lowered its latency impact by snooping only when packets switch dimensions (*e.g.* from the X to the Y dimension)

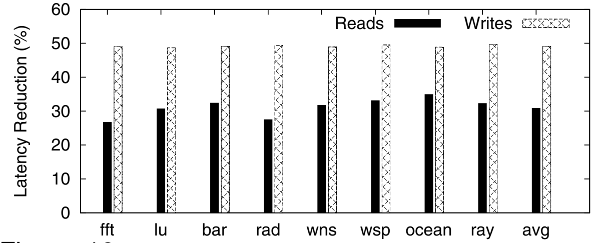



Figure 10. Effect of integrating the routing protocol within the network as opposed to within the network *interface* in the local node.

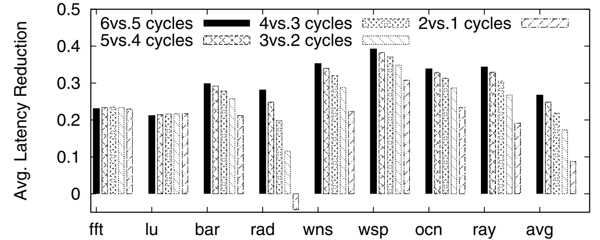



Figure 11. Average memory access reduction as a function of router pipeline depth.

### 3.8. Potential of network-level optimizations

Figure 11 shows the overall memory access latency reduction when the router pipeline of the baseline MSI protocol is varied from 1 to 5 cycles. For each point of comparison, we assume that our protocol has an extra cycle of latency (so the comparison to the 1-cycle directory protocol pipeline is a 2-cycle pipeline for our protocol). We see that as the pipeline is shortened, the overall performance improvement lessens; but this is to be expected since as each pipeline is reduced by one cycle, the *ratio* of the two pipeline lengths increases (*e.g.* $2/1 > 6/5$). It should be noted that while techniques such as speculative allocators [18] can shorten pipelines, they only work at very low network loads, lengthening the router pipeline instead at moderate to high loads when speculation frequently fails.

## 4. Related Work

There has been extensive prior research in the area of cache coherence. Here, we attempt to classify the most relevant prior research into three groups: those that jointly explore the design of cache coherence protocols and the network fabric; those that optimize the communication network for cache coherence; and those that propose new protocol optimizations that are oblivious to the underlying network fabric.

**Joint protocol-network optimizations.** Several prior works have also jointly optimized the cache coherence protocol and the network. Barroso and Dubois [34] adapted the snooping protocol that was thought to be only applicable to buses so it works on ring networks as well. Cheng *et al.* [35] leveraged the heterogeneous interconnects available in the upper metal layers of a chip multiprocessor, mapping different coherence protocol messages onto wires of different

widths and thicknesses, trading off their latency-bandwidth requirements.

The first work to implement cache coherence in the network layer was that by Mizrahi *et al.* [36]. In their work, the entire data cache is migrated into the network routers. In the domain of on-chip networks, it is not feasible to cache actual data within the network fabric as the access time will critically affect the router pipeline. Our proposal decouples the coherence protocol from data storage. Furthermore, since the protocol assumed was one in which only one copy of any given data exists in the network at any time, it is trivially coherent by definition. On the other hand, we require explicit tracking of sharers, and propose in-network directories (virtual trees) for this purpose.

Like our proposed in-network implementation, the Wisconsin Multicube [37] enables in-transit "snarfing" of cache lines, made possible by its grid-of-buses topology where each cache monitors two broadcast buses – a row bus and a column bus. However, while the use of buses enabled in-transit snooping and snarfing, it also results in the need for invalidates to be fully broadcasted. Besides, buses are used solely for communications, with directory tables keeping track of modified lines kept in the caches, above the network. By moving directories into the network, we show that a point-to-point network can efficiently facilitate the in-transit snooping capabilities of a broadcast medium.

The closest work to our proposed in-network cache coherence protocol is GLOW by Kaxiras and Goodman [17]. They propose similarly mapping trees connecting sharers to the underlying network topology, so a sharer who is closer by than the directory can be reached to reduce latency. However, they stop short of embedding the directories fully into the network, instead implementing the protocol at network interface cards. The underlying network thus remains purely for communications. They also retain the original bit-vector directories at the home nodes. By implementing the directories as virtual trees within network routers, we can realize very-low-latency steering of the coherence traffic. As Section 3 shows, an in-network implementation of directory-based MSI results in an average 40% delay savings over an above-network implementation.

**Network optimizations for cache coherence.** Prior work has demonstrated the impact of the interconnection network on overall shared-memory system performance and proposed network designs that can better the performance of cache coherence protocols. Stets *et al.* [19] demonstrated that network features, such as network ordering, multicasting, can greatly simplify coherence protocol implementation and improve system performance. Dai and Panda [20] presented block correlated FIFO channels to tackle the memory ordering issue of network interface design. They also proposed a multi-destination message-passing approach to lower the cache invalidation overhead [21]. Bilir *et al.* [22] designed multicast snooping to minimize communication traffic. The cruise-missile-invalidates of the Piranha machine [23] applies multicasting to invalidates so a few invalidation messages can be used to invalidate a large number of nodes. While these works tailor the network for better performance of cache coherence protocols, the network is only responsible for delivering a message to the specified destination. By embedding directories *within* the network in the form of virtual trees, we enable in-transit rerouting of messages towards sharers of the cache line, *away* from the originally specified destination (the home node).

**Protocol optimizations.** It has been shown that it is expensive to enforce sequential consistency [24], as it requires ensuring a strict ordering between read/write accesses that mandates multiple round-trips of request-invalidate-acknowledgment-reply communications. As a result, extensive prior research has focused on optimizing the delay of sequentially consistent systems, through new protocol states, with protocols supporting relaxed consistency the most prevalent [24]. DASH [25] combines both a snooping bus-based protocol with a directory-based protocol through a two-level interconnect hierarchy. DASH supports efficient execution of read-modify-write accesses because a clean exclusive cache line can be replaced without notifying the directory. Later on, this protocol was adapted in SGI Origin systems [26]. Shen *et al.* [27] proposed CACHET, an adaptive cache coherence protocol. To minimize write latency, CACHET allows stores to be performed without the exclusive ownership, which enables concurrent writes accessing the same address. Huh *et al.* proposed speculative incoherent cache protocols [28]. Coherence decoupling breaks communication into speculative data acquisition and coherence validation with rollback upon misspeculation. While relaxing the consistency model lowers communication overhead, it also complicates the programming model. Our proposed in-network implementation of the MSI protocol reduces communication overhead while ensuring sequential consistency.

Furthermore, end-to-end protocol optimizations are orthogonal to in-transit optimizations. For instance, our in-network virtual trees can be used to percolate tokens of the TokenB coherence protocol [29] efficiently within the network, between sharer nodes. They also provide a natural way of replicating cache lines from the root node towards requestors dynamically depending on available cache capacity, reducing average access latency, demonstrating how end-to-end replication protocols [11] can be moved into the network for better efficiency. In short, while we demonstrate in this paper just the efficiency of an in-network implementation for the widely used directory-based MSI protocol, we believe the implementation of many protocols within the network will reap latency benefits as it allows protocols to very efficiently leverage the physical locality exposed by the network.

Substantial prior work tackled the storage scalability issues of full-map directory-based protocols. Variants of limited directories [15] were proposed, from storing pointers to a subset of sharers, to coarse vectors, compressed and multi-level directories [30, 31]. In these techniques, in general, a subset of pointers to sharers are kept at the directory node, and pointer overflows are handled by limited broadcasts, second-level directories or by software. Alternatively, coherence protocols were proposed where directories are maintained as linked lists (such as SCI [16]) or trees (such as [32, 33]). Unlike limited directory schemes, these protocols track all sharers and thus do not trade off fidelity. However, pointers take up $O(logN)$ storage, where $N$ is the number of nodes in the network, and a parent-child relationship in the list or tree does not necessarily map to a neighboring tile relationship in the physical network. Therefore, the number of actual hops required for a message to traverse the list/tree from the root to any of the leaves is bounded from above by $DlogN$, where $D$ is the diameter of the network. Like full-map directory protocols, our in-network cache coherence keeps track of *all* sharers. Similarly to list/tree protocols, we achieve this through pointers, though again by moving them into the network. By embedding these directory pointers at each router, pointers take up $O(logP)$ storage, where $P$ is the number of ports or degree of the network, and physical network locality can be leveraged ef-

ficiently since the pointers can be accessed directly within each router, without leaving the network fabric.

## 5. Conclusions

In this paper, we propose the embedding of cache coherence protocols within the network, separately from the data they manage, so as to leverage the inherent performance and storage scalability of on-chip networks. While there has been abundant prior work on network optimizations for cache coherence protocols, to date, most prior protocols have maintained a strict abstraction of the network as a communication fabric. Here we detailed how the directories of classic directory-based protocols can be moved into the network, maintained in the form of virtual trees that steer read and write requests in-transit, towards nearby copies. Our evaluations on a range of SPLASH-2 benchmarks demonstrate up to 44.5% savings in average memory latency on a 16-processor system. Furthermore, the performance scalability is demonstrated by an average memory access savings of up to 56% savings on a 64-processor system. Ultimately, we envision the embedding of more distributed coordination functions within the on-chip network, leveraging the network's inherent scalability to realize high-performance highly-concurrent chips of the future.

## Acknowledgments

The authors would like to thank Margaret Martonosi of Princeton University and David Wood of The University of Wisconsin-Madison for valuable comments on this work. We also wish to thank the Princeton NDP group, especially Kevin Ko, for help with the benchmarks. This work was supported in part by MARCO Gigascale Systems Research Center and NSF CNS-0509402, as well as NSERC Discovery Grant No. 388694-01.

## References

[1] http://www-128.ibm.com/developerworks/power/library/pa-expert1.html.
[2] http://www.intel.com/multi-core/.
[3] http://www.sun.com/processors/throughput/.
[4] "International technology roadmap for semiconductors," http://public.itrs.net.
[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.
[6] D. L. Dill, "The mur$\phi$ verification system." in *Proc. 8th Int. Conf. Comp. Aided Verif.*, Aug. 1996, pp. 390–393.
[7] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA: Morgan Kaufmann Publishers, 2003.
[8] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. on Comp.*, vol. c-28, no. 9, pp. 690–691, Sept. 1979.
[9] http://www-flash.stanford.edu/apps/SPLASH/.
[10] K. P. Lawton, "Bochs: A portable pc emulator for unix/x," *Linux J.*, vol. 1996, no. 29es, p. 7, 1996.
[11] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *Proc. 32nd Int. Symp. Comp. Arch.*, Jun. 2005, pp. 336–345.
[12] S. Mukherjee, *et al.*, "The Alpha 21364 network architecture," in *Proc. Hot Interconnects 9*, Aug. 2001.
[13] S. J. Wilton and N. P. Jouppi, "An enhanced access and cycle time model for on-chip caches," DEC Western Research Laboratory, Tech. Rep. 93/5, 1994.
[14] M. B. Taylor *et al.*, "The RAW microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE-MICRO*, vol. 22, no. 2, pp. 25–35, Mar./Apr. 2002.
[15] A. Agarwal *et al.*, "An evaluation of directory schemes for cache coherence," in *Proc. 15th Int. Symp. Comp. Arch.*, Jun. 1988, pp. 280–289.
[16] S. Gjessing, *et al.*, "The SCI cache coherence protocol," Kluwer Academic Publishers, 1992.
[17] S. Kaxiras and J. R. Goodman, "The glow cache coherence protocol extensions for widely shared data," in *Proc. 10th int. conf. Supercomputing*, May 1996, pp. 35–43.
[18] L.-S. Peh and W. J. Dally, "A delay model and speculative architecture for pipelined routers," in *Proc. 7th Int. Symp. High Perf. Comp. Arch.*, Jan. 2001, pp. 255–266.
[19] R. Stets, *et al.*, "The effect of network total order, broadcast, and remote-write capability on network-based shared memory computing," in *Proc. 6th Int. Symp. High Perf. Comp. Arch.*, Feb. 2000, pp. 265–276.
[20] D. Dai and D. K. Panda, "Exploiting the benefits of multiple-path network in DSM systems: Architectural alternatives and performance evaluation," *IEEE Trans. Comp.*, vol. 48, no. 2, pp. 236–244, 1999.
[21] D. Dai and D. Panda, "Reducing cache invalidation overheads in wormhole routed DSMs using multidestination message passing," in *Proc. 1996 Int. Conf. Par. Processing*, Aug. 1996, pp. 138–145.
[22] E. E. Bilir, *et al.*, "Multicast snooping: a new coherence method using a multicast address network," in *Proc. 26th Int. Symp. Comp. Arch.*, Jun. 1999, pp. 294–304.
[23] L. Barroso *et al.*, "Piranha: A scalable architecture based on single-chip multiprocessing," in *Proc. 27th Int. Symp. Comp. Arch.*, Jun. 2000, pp. 282–293.
[24] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, 1996.
[25] D. Lenoski, *et al.*, "The DASH prototype: implementation and performance," *SIGARCH Comp. Arch. News*, vol. 20, no. 2, pp. 92–103, 1992.
[26] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA highly scalable server," in *Proc. 24th Int. Symp. Comp. Arch.*, Jun. 1997, pp. 241–251.
[27] X. Shen, Arvind, and L. Rudolph, "CACHET: an adaptive cache coherence protocol for distributed shared-memory systems," in *Proc. 13th Int. Conf. Supercomputing*, Jun. 1999, pp. 135–144.
[28] J. Huh, *et al.*, "Speculative incoherent cache protocols," *IEEE Micro*, vol. 24, no. 6, Nov./Dec. 2004.
[29] M. M. K. Martin, M. D. Hill, and D. A. Wood, "Token coherence: Decoupling performance and correctness," in *Proc. 30th Int. Symp. Comp. Arch.*, Jun. 2003, pp. 182–193.
[30] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "Limitless directories: A scalable cache coherence scheme," in *Proc. 4th Int. Conf. on Arch. Support for Prog. Lang. and Op. Sys.*, Jun. 1991, pp. 224–234.
[31] M. E. Acacio, *et al.*, "A new scalable directory architecture for large-scale multiprocessors," in *Proc. 7th Int. Symp. High Perf. Comp. Arch.*, Jan. 2001, pp. 97–106.
[32] H. Nilsson and P. Stenström, "The Scalable Tree Protocol – A Cache Coherence Approach for Large-Scale Multiprocessors," in *Proc. 4th IEEE Symp. Par. and Dist. Processing*, Dec. 1992, pp. 498–506.
[33] Y.-C. Maa, D. K. Pradhan, and D. Thiebaut, "Two economical directory schemes for large-scale cache coherent multiprocessors," *SIGARCH Comp. Arch. News*, vol. 19, no. 5, p. 10, 1991.
[34] L. Barroso and M. Dubois, "Performance evaluation of the slotted ring multiprocessor," in *IEEE Trans. Comp.*, July 1995, pp. 878–890.
[35] L. Cheng, *et al.*, "Interconnect-aware coherence protocols," in *Proc. 33rd Int. Symp. Comp. Arch.*, Jun. 2006, pp. 339–351.
[36] H. E. Mizrahi, *et al.*, "Introducing memory into the switch elements of multiprocessor interconnection networks," in *Proc. 16th Int. Symp. Comp. Arch.*, Jun. 1989, pp. 158–166.
[37] J. R. Goodman and P. J. Woest, "The wisconsin multicube: a new large-scale cache-coherent multiprocessor," in *Proc. 15th Int. Symp. Comp. Arch.*, Jun. 1988, pp. 422–431.