MESIF: A Two-Hop Cache Coherency Protocol for Point-to-Point Interconnects (2004)

J.R. Goodman
University of Auckland

&

H.H.J. Hum
Intel Corporation

Abstract–*We describe MESIF, a new cache coherence protocol. Based on point-to-point communication links, the protocol maintains no directory, and mimics the broadcast behavior of a snooping cache protocol. MESIF supplies data cached in other nodes in a single round-trip delay (2-hop latency) for all common operations. Because of the speed of the links, the protocol can outperform a bus-based protocol for a small number of nodes, but scales through hierarchical extension to a large-scale multiprocessor system. Salient features of the industrial-quality protocol are described. The introduction of a novel forwarding state is used to assure a single response to shared data and to simplify conflict resolution. In the hierarchical extension, auxiliary hardware data structures can be included to provide 2-hop latency for most operations.*

The attached manuscript was submitted on 19 November 2004 to be considered for inclusion in the *27th International Symposium on Computer Architecture (ISCA'05)*. To preserve its historical authenticity, it is presented verbatim (including known errors).

The MESIF protocol described was the first of a collection of MESIF protocols developed at Intel Corporation. A forerunner of the QPI protocol subsequently developed by Intel, it was first proposed in 2001 by Herbert Hum, an Intel employee, and James Goodman, a consultant on sabbatical from the University of Wisconsin-Madison.

MESIF: A Two-Hop Cache Coherency Protocol for Point-to-Point Interconnects

Abstract—*We describe* MESIF*, a new cache coherence protocol. Based on point-to-point communication links, the protocol maintains no directory, and mimics the broadcast behavior of a snooping cache protoc9l. MESIF supplies data cached in other nodes in a single round-trip delay (2-hop latency) for all common operations. Because of the speed of the links, the protocol can outperform a bus-based protocol for a small number of nodes, but scales through hierarchical extension to a large-scale multiprocessor system. Salient features of the industrial-quality protocol are described. The introduction of a novel forwarding state is used to assure a single response to shared data and to simplify conflict resolution. In the hierarchical extension, auxiliary hardware data structures can be included to provide 2-hop latency for most operations.*

1. Introduction

There are two well-known categories of cache coherence protocols: directories and snooping caches. Snooping cache coherence protocols distribute the coherence information around the system, making cache controllers responsible for maintaining information about the whereabouts of copies of the memory locations for which they have special privileges. They depend on a broadcast to assure that all the nodes witness events that might require their intervention. While this idea has been extended with the use of a logical bus[1], the basic idea of a broadcast is fundamental to the snooping protocol.

Directory-based schemes depend on a single place—the directory—to store information about a cache line. While this information may be distributed for different addresses, each address is associated with a given location where the coherence information is maintained (or at least, where it can be found[2, 3]). Unfortunately, this means that many common operations require a minimum of three serial communications to complete a memory transaction, as opposed to two for a snooping protocol.

Emerging technologies and energy considerations are making broadcast through a common electrical medium less attractive. Point-to-point communications have the potential for both lower latency and higher bandwidth communication[4]. We have investigated the possibility of "broadcasting" through parallel, point-to-point links, achieving the 2-hop latency of snooping protocols while exploiting the higher potential bandwidth of point-to-point links. We have achieved this with the MESIF protocol, which provides a 2-hop latency for all the common

memory operations. The authors are aware of two commercial implementations of non-directory-based cache coherency protocols using point-to-point interconnects: AMD™'s coherent HyperTransport™[5] and Intel™'s Scalability Port™[6][1]. However, both require a minimum of three hops for common memory operations.

Because of the complete interconnect required, the size of a cluster of nodes so connected is technology-dependent, but clearly does not scale to a large number of nodes. The MESIF protocol scales by introducing a hierarchical model that implements the 2-hop protocol at multiple levels. Despite the fact that some operations may require multiple hops over different levels of the hierarchy, this latency can largely be overcome by the use of caches to maintain critical coherence information at critical points in the network, meaning that most common memory operations can still be performed with only two hops. Thus MESIF can support higher performance than a snooping protocol for a small number of nodes, but is capable of scaling to a large number of nodes, like a directory-based scheme.

The protocol is designed to exploit the fact that shared data can be retrieved from another cache faster than it can be fetched from memory. Shared data, as well as modified data, is supplied out of a cache rather than fetching it from memory whenever possible.

The goals of the MESIF protocol are the following:

1. High efficiency for a small number of nodes,
2. Two-hop protocol for common memory operations within a leaf cluster,
3. No back-off/retry,
4. Maximize throughput by handling concurrent requests efficiently,
5. Scale to large system, without need for directory protocol,
6. Scalable in a hierarchical manner, and
7. Independent of switching fabric.

An interconnect that provides broadcast capability offers an important advantage: serial observability. The serial nature of a bus provides a global order in which events are observed, even if they are observed at slightly different times. But even if messages along each link are delivered in-order, point-to-point interconnects suffer from "time-warp" (see Fig. 1). MESIF recaptures most of the properties of a bus by requiring a sequence of

[1] All trademarks are the properties of their respective owners.

responses, assuring that individual messages, even if delayed, will not prevent the system from achieving serial observability. The protocol includes the concept of a transaction that begins with a broadcast, where a node sends a (nearly identical) message to each of the other nodes, requesting privileges, status, and/or data. Because broadcasts may overlap, the transaction is not ordered until data is received, or at least one response has been received from each of the other nodes (in some cases more than a single response may be required). Request/response assures that after a period of uncertainty, any pair of nodes concurrently attempting transactions on the same cache line will both be aware of a conflict. The MESIF protocol guarantees that a conflict will be resolved quickly and handled efficiently. While the winner is determined immediately, the Home node (the node containing main memory for that cache line) is responsible for assuring that the loser's request is queued and handled efficiently and fairly. The scheme naturally extends to multiple concurrent requests.

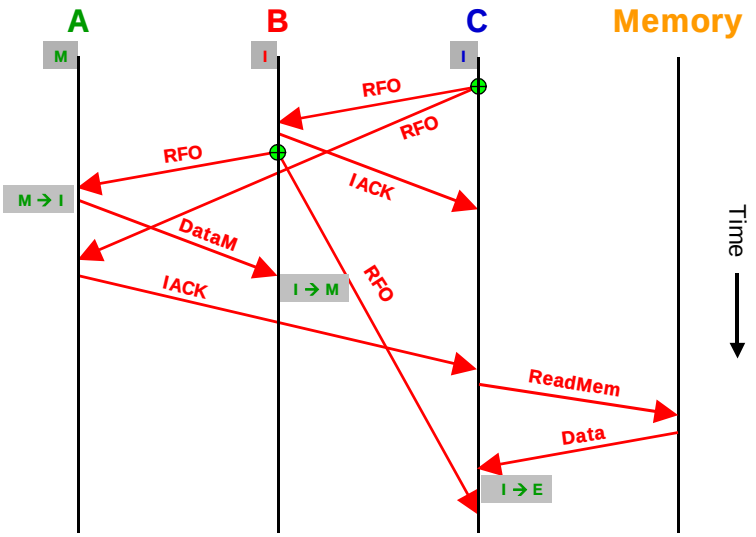


Fig. 1. Time Warp. Node C first initiates a Read-for-ownership (RFO), sending requests to A and B. Node B responds to the RFO, indicating it does not have the line, later initiating its own RFO for the same line. Node A receives the RFO from B first. Having the data in Modified state, it responds to the request by passing the data to B and invalidates its own copy. Now when it receives RFO from C, it also responds that it does not have the line. C now requests the cache line from memory, having verified that neither A nor B have it. But B has the copy sent from A, creating two incompatible copies.

2. Protocol Description

2.1. The Basic MESIF Protocol

A cluster consists of a collection of nodes. Each node may have either a cache controller (with attached processor), or main memory, or both. For every cache line address, one node (called Home) contains the main memory for that address. These two functions (memory and cache) are independent, but co-exist in the node. In the hierarchical model, an entire cluster of nodes consisting of processors, caches, and memory will be treated as single node that consists of the same two entities. We use the term "Home" to refer to the memory and supporting controller. We sometimes use the term "node" to refer only to the cache controller, using "Home node" if we mean to speak of both. Because the functions can be cleanly separated, we may in some cases refer to Home as if it did not have cached data, and we may refer to "all nodes" as if one were special, that is, the Home node for the transaction.

In general, messages are directed to one entity or the other. For the messages in a typical transaction, specifically, the first (broadcast) message is sent to the node (cache controller), not Home, while the third is directed to Home, not the node. Home does not respond to the broadcast message, though it may use it to initiate a speculative fetch. The protocol names imply that Home does see the initial request: the later message is sent to Home either confirming or canceling the original request. If the Home node has no caching capability, the protocol can be optimized slightly: Home may be sent the initial broadcast message (or not), but it does not respond until requested in a subsequent phase.

A cache line may be in one of five states in each of the nodes: M (Modified), E (Exclusive), S (Shared), I (Invalid), or F (Forwarding). The first four states correspond to the classic states of a MESI protocol[7]. The Forwarding state indicates a "first-among-equals," that is, a state assigned to a single node among those sharing a cache line, and responsible for providing it when a request is received. This state is similar to an "Owned" state, but coherent with memory and useful for read requests, not just write requests. The Forwarding state in MESIF primarily facilitates the rapid response of a cached copy (requestor broadcasts and responder provides a cached copy for a 2-hop latency) in the presence of multiple cached copies. The F state also simplifies the conflict resolution mechanism in the protocol. This will become evident in a later section.

A transaction typically consists of five sets of messages, which may be partially overlapped. In many cases, one of the first response messages returns data, so the requesting node can quickly acquire and (provisionally, depending on the memory model) use the data before the transaction is completed. For all memory models, the data can be used after the fourth message.

A transaction is initiated by a broadcast to all nodes, including the Home node. All nodes must respond. A response may include data (from at most one node), and indicates:

1) The state of the cache line in the responding cache, and
2) An indication if a conflicting request has been detected.

In order to guarantee the serializability of transactions, a node supplying data may not respond to any further requests for the same cache line until it has received acknowledgement from the requesting node. The requesting node may use the data, but it is not allowed to acknowledge receipt from the sender until it receives acknowledgement from Home.

After responses have been received from all nodes, a second message (second phase) is sent to Home. This message, either a READ or CNCL requests data (a READ) or indicates that it has received data (CNCL). In either case, the message must also enumerate all conflicting requests it has detected for this cache line between the time it initiated the request and the time it received the last response. If a conflict has been detected, the node signals either that it was the winner (CNCL) or a loser (READ).

Home must respond to a READ request by sending either the data from memory or instructing another node to forward the data (for conflict cases). In the case of a conflict, Home must identify all parties to the conflict and provide for the serial transfer of the cache line to each of the requesting nodes. It does this by sending forwarding messages to the winner and all but one of the losing nodes in the conflict, also informing each losing node to await data from a specified node. If there is no conflict, Home responds to a CNCL by acknowledging (ACK) receipt of the message and thus terminating the transaction.

A transaction may be initiated by any of the following requests.

1) ReadShared (RS)
2) Read for Ownership (RFO)
3) Write back (dirty eviction)
4) Invalidation request (upgrade to ownership[2])

One of the following responses will be returned from each node except Home:

[2] There are many other transaction types such as requests for handling data crossing cache lines, non-coherent transactions, etc. that are present in a production-quality protocol.

1) IACK: invalid state acknowledgement,
2) SACK: shared stated acknowledgement (no data sent),
3) Data&State: data sent along with state (F, E, or M) to transition to, and
4) Conflict: there is already a pending request for the same cache line.

The Home is permitted to respond immediately upon receiving a READ or CNCL message from the requestor. It must not wait for all conflicting requests to arrive, because there are cases where some cannot be generated until it has responded.

We make no attempt to describe the complete protocol in this paper. There are many subtle cases, too complex to describe completely in a document at this level[3]. Here the goal is simply to convey the general nature of the protocol. Though we have confidence in the correctness and completeness of the protocol (formal validation efforts on multiple variants of this protocol have successfully been completed), the details are yet be released.

2.1.1. Transaction Flow for a Read Request

We start by describing a non-exclusive read request. A requesting node broadcasts to all peer nodes a request and waits for their responses. There are three possible cases. For the moment we ignore the conflicts—overlapping requests for the same cache lines. Requests for different cache lines can be arbitrarily overlapped.

i.) The requested line is uncached.

All nodes except the Home node respond with a IACK (Invalid copy ACKnowlegement). After all IACKs have been received, the node sends a confirming READ request to Home. Home then responds with the data, completing the transaction. The node assigns the E state to the cache line.

ii) The requested line exists in one or more nodes in state S, with at most one node in state F.

(There may not be an F copy, but rather S copies. In this case, the requesting node reads the data from Home and receives it in state F). If the line exists with state F in some node, that node forwards a copy of the line to the requesting cache, changing its own state to S. The receiving cache acquires the new line in state F and the

[3] For instance, a processor may require self snooping, memory types can be uncacheable, etc. All of these cases need to be handled in a production-worthy cache coherency protocol, but are too detailed to discuss in this paper.

requestor can effectively use the data. After responses have been received by all nodes except Home, the requesting node sends a cancel message to Home. Upon receiving an acknowledgement message from Home, a DACK message is sent from the requestor to the sending node, completing the transaction (see fig. 2). The DACK message frees the old owner to respond to other requests for the same line. (More will be described in a later section detailing the conflict cases.)

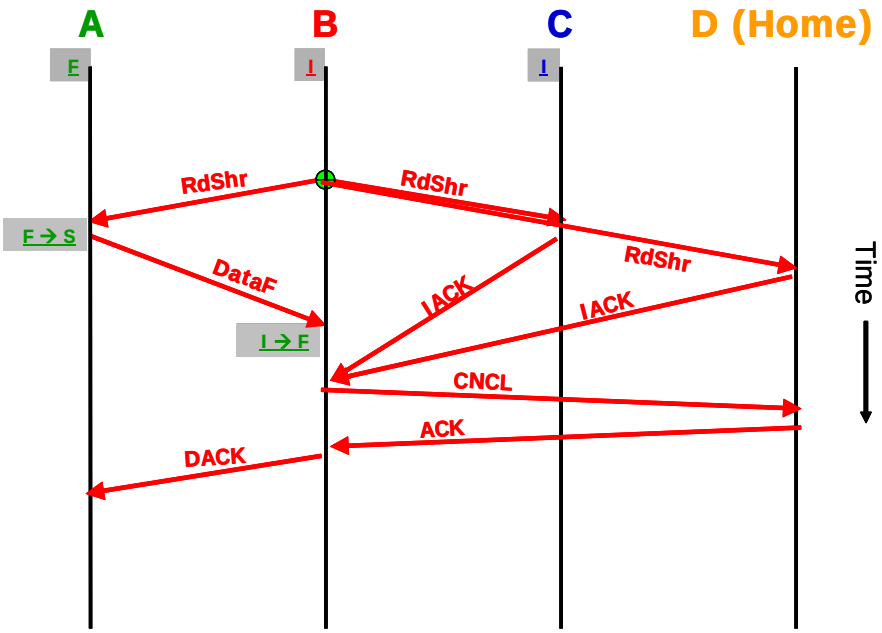


Figure 2. A ReadShared transaction flow with a cached F copy.

The F state allows the current owner, which may not be the home node, to determine who the new owner for the next request (to the same cache line) is without any intervention on the part of the home node—assuming that the next request has no conflict. In this manner, multiple nodes can serve as proxies for the home node (in deciding a new owner) on simple, uncontended requests—the vast majority of requests.

iii) The requested line exists in a single node in E state.

The case is treated exactly the same as if the cache line were in F state instead of E.

iii.) The requested line exists in a single node in M state.

This case is tricky because the state has been requested for reading shared but the data is modified. In the basic protocol, the node is transmitted to the requesting node as if the request had been a Read for Ownership(RFO). A variant allows the data to be shared, allowing the responding node to keep a copy in state S. The requesting

node now must "cleanse" the line by writing it back to memory, possibly as a variant of the CNCL request. Alternatively, it may maintain the cache line in a Shared-Modified state (FM), deferring the requirement to cleanse the line. This latter choice, while seeming to complicate the protocol, is actually the easiest to implement.

2.1.2. Transaction Flow for a Read For Ownership Request

A read request with permission to modify, otherwise known as a Read for Ownership (RFO) is handled similarly to a read request. Again there are three possible cases.

i.) The requested line is uncached.

This case is handled exactly like a read request.

ii) The requested line exists in a single node in M or E state.

This case is handled exactly like a read request, except that the delivered data is purged from the sending cache and the state of the received line does not change. Once the requestor receives the M or E cache line, the local processor can commence modifying the data (In x86 bus parlance, this is called the Global Observation point). However, the modified data cannot be shared with the rest of the system until the Home acknowledges that it is the new owner (through an ACK message or a forwarding (XFR) message; the latter effectively exposing the modified data to the rest of the system).

iii) The requested line exists in one or more nodes in state S, with at most one node in state F.

This case is handled like a read request except that each node invalidates its cache line before responding (they respond with an IACK). This includes the node in the F state, if one exists. Obviously, the node with F state responds with the data and replies with an IACK message to the requesting node.

2.1.3. Transaction Flow for Invalidate Line Requests

A request for permission to modify a shared line already in the cache of the requesting node is similar to the RFO request, except that no data is transferred (this is called an Invalidation request). Regardless of the state of the cache line elsewhere, the requesting cache broadcasts the invalidation request. After receiving responses from all nodes indicating they have purged any copy they have, the requesting node confirms the request to the Home node with a CNCL message, awaiting an acknowledgement to complete the transaction. In the case where

there is a conflict, i.e. another request snoops the cache line out under the requestor, the Invalidate Line request gets morphed by the Home and the Home sequences a transfer of data from the owner to the requestor.

2.1.4. Transaction Flow for Writeback Line Requests

When a modified cache line is evicted, a Writeback request is issued from the owner node to the home node. During this transaction, the ownership is effectively transferred back to the home node and while this transaction is pending, the node performing the writeback buffers all snoops to the cache line until the home node responds with an acknowledgement. At that point, the node can respond with an IACK to any incoming snoops for that cache line (see fig. 3).

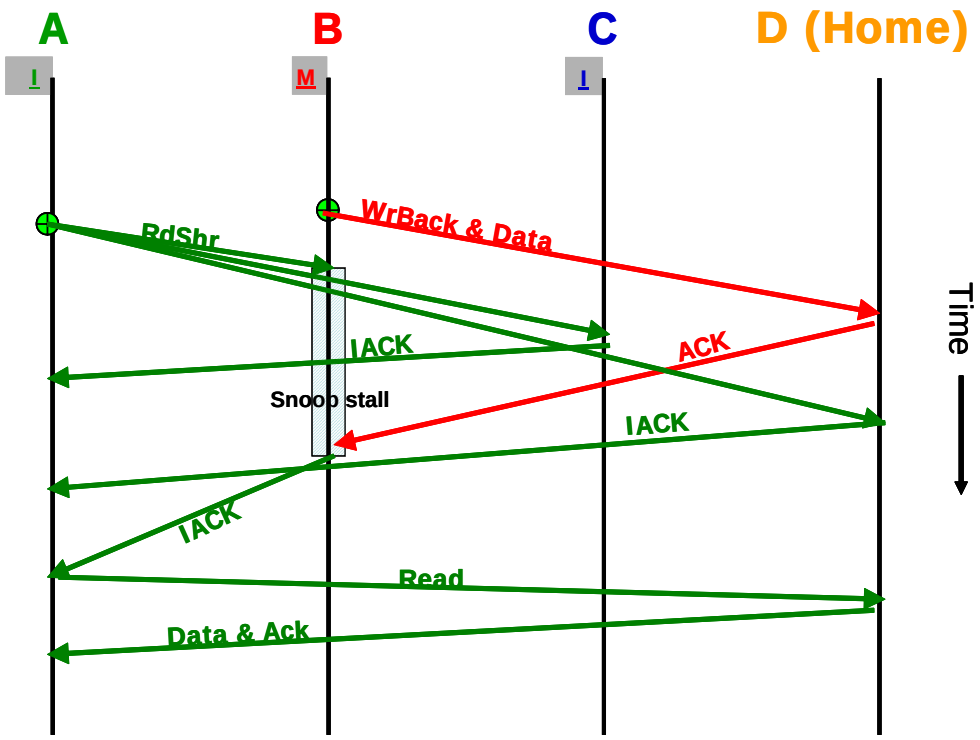


Figure 3. Writeback transaction stalling snoops to the same cache line[4].

2.2. Dealing with Conflicts in MESIF

The concurrency inherent in a point-to-point network means that two or more requests may overlap. If the requests are to different cache lines, the protocol guarantees a global order by establishing the point at which all copies of a cache line (other than the one being modified) have been eliminated. Two requests to the same

[4] This example shows D as a Home node, i.e. it contains both the memory controller owning the cache line plus a caching agent.

cache line, however, require some sophistication to assure consistent state and correct global ordering. The Home plays a special role in every transaction not only to supply data that is not cached, but also to make sure that conflicting transactions to the same cache line are handled correctly and efficiently.

If two nodes, A and B, generate requests to the same cache line simultaneously, the order in which they are observed at other nodes may be different. Furthermore, at least one, and possibly both requesting nodes will observe an incoming request to occur *after* the local request is generated. This is guaranteed because it is impossible for each node to handle a concurrent remote request before it generates its own. Thus for any conflict, at least one of the nodes will detect that it has an outstanding request at the time a conflicting request is received.

Usually a conflict is quickly resolved by the node that supplies the data. The F state simplifies conflict resolution by determining a single node to respond to requests for data that is shared. (If the data is cached in E or M state, a single node plays the same role.) The node resolves the conflict by sending the data to one of the two nodes (presumably, but not necessarily, to the node whose request is received first). This simple resolution is quick, but it does not guarantee that the loser will eventually receive the data. A possible solution is simply for the loser to try again. For a variety of reasons we have rejected that solution, requiring that the Home also be involved in all conflicts, directing the winner to forward the cache line to the loser. This is necessary even in the case of read-only (shared) data, because the node supplying the data initially (in F state) passes on the F state property to the winning requester. Allowing the node with the F-state to supply the cache line to two nodes might seem more efficient, but it greatly complicates the protocol, which requires no more than one response to each request for data, and because it might result in multiple nodes with a cache line in the F state. Figure 4 gives an example of a simple conflict between two nodes simultaneously requesting ReadShared data.

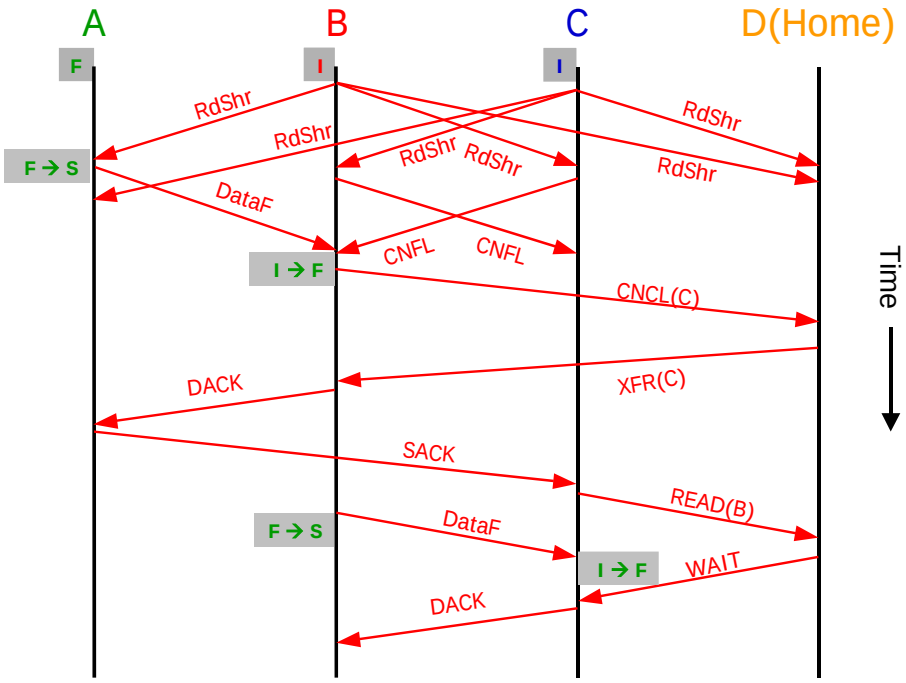


Fig. 4. Conflict resolution[5]. Nodes B and C simultaneously generate ReadSharing requests (RdShr) for a cache line that is present only in node A. Node B's request is received first by node A, which responds by sending the line (DataF) to B. Because it has supplied data in a transaction that is not completed, node A is not permitted to respond to the RdShr request from node C and must snoop stall until it receives acknowledgement from B. Both nodes B and C respond to the others request by indicating a (CNFL). The winner, B, proceeds by canceling its request to Home and indicating the conflict with B (CNCL(B)). Home responds to the cancel request with conflict by instructing node B to forward the line to node C (XFR(C)), After receiving the response from Home, node B acknowledges the data received from A (DACK), and after accessing it momentarily, forwards the data to B (DataF). After receiving the response from A (SACK) but no data, node C now requests data from the Home node, indicating its conflict with B (READ(B)). Home has already instructed node B to forward the data to node C(XFR(C)), and node B may have already delivered the data to C by the time it receives the response from Home (WAIT)[6]. After receiving both signals, node C acknowledged receiving the data from node B (DACK),

Because of timing delays, only one of two conflicting nodes may actually detect the conflict. The protocol is robust enough to handle these cases. The Home node is able to recognize and disambiguate all conflicts from the READ or CNCL it receives from each requesting node (indicating all detected conflicts). This is true even though some nodes will not be aware of a conflict and simply ask for data or cancel their request (because it has been satisfied).

[5] This example shows D as simply as Home, i.e. without a caching agent.
[6] The WAIT message is equivalent to an ACK message, but is separately identified for ease of description.

Note tat th old owner in figure 4 (node A) stalls all pending snoops to the same cache line during the period it sent off the cache line until it receives the DACK from the new owner. This is to allow the winner to notify the Home. When the loser sends its second request to Home, the conflict will be detected and captured in the conflict list that is sent to Home.

2.2.1 Different Kinds of Requests May Conflict

A ReadShared request and an RFO request may occur concurrently. The protocol simply handles them in order, with the transfer assuring that the successful RFO will result in the invalidation of all copies but one. In some cases, a ReadShared request may also result in a modified (exclusive) copy being supplied.

2.2.2. More Than Two Nodes May Conflict Generating Overlapping Requests

In general, a conflict is detected whenever a node detects a conflicting request from another node after it has generated its own request and before it has received its last response. Once a conflict is detected, Home is responsible for sorting out the winner and determining the order for all losers. Because of snoop stalls, additional conflicts may arise once the forwarding mode has begun and can continue indefinitely for a heavily shared line. While the logic to handle this case is somewhat complex for the Home node, the sequence of cache line transfers is nearly optimal. Such heavily shared data are sometimes critical to an application and handling them efficiently is highly desirable

The protocol handles multiple conflicts in a natural extension to the simple conflict: each node other than the winner is assigned a position in the queue (that is kept in the Home node), and the Home node sends messages to each of the competing nodes. The messages from Home consist of two pieces of information:

1) The address of the node to which it should forward the line, and
2) Notification to wait for the cache line that will eventually be forwarded to it.

Obviously Home doesn't send the first message to the very last node, and it doesn't send the second message to the winner, which receives the data in the normal way.

The conflict resolution protocol outlined in this paper has been formally validated.

2.3. Virtual Channels for Deadlock Avoidance

Since different type messages flow in both directions between any two nodes, care must be taken to avoid deadlock. In MESIF, three virtual channels for each direction are required for deadlock avoidance:

Channel 1: initial requests such as RdShr, RFO, etc.

Channel 2: snoop responses such as IACK, SACK, and commands to Home, READ, CNCL, etc.

Channel 3: DACK, XFR from Home

Data can be transferred on channel 3 if all data can be buffered at the destination. The way to reason about virtual channels is that all message types terminating a transaction must be grouped together and must flow unimpeded to the requestor (channel 3). Responses to requests must be buffered at the destination and must not be blocked by initial requests (channel 2). Lastly, initial requests can be stalled due to resource limitations and must be kept on a separate channel (channel 1).

2.4. Hierarchical MESIF

For any line that is cached, the MESIF protocol provides the data to a requesting node in a single roundtrip delay for the simple protocol. If the data is uncached, there is additional delay, because the requesting node must send a second request to the Home node after it has received a response from all other nodes. This time can presumably be overlapped with the memory access, and is equivalent to the delay assumed in a snooping cache protocol that allows cache controllers on the bus to intervene, preventing memory from supplying d

The limitation of the MESIF protocol, however, is the requirement for a complete interconnect among the nodes for efficient broadcasting. Thus there is a hard limit in the size of the basic MESIF cluster, determined by the number of links from each node. The MESIF protocol can be extended beyond a single cluster by introducing the notion of a pair of *agents* for the cluster for each address. These agents represent the rest of the system to the cluster, and represent the cluster as if it were a single node to the rest of the system. A simple cluster can be thought of as a leaf in a tree, where agents attached to the cluster act as nodes in a cluster in the next level of the tree. A multi-level tree is envisioned by composing clusters of clusters, etc.

Agents recognize requests that must be handled—or at least propagated—to other clusters. From the standpoint of a cluster, a *Home agent* behaves exactly like the Home node for all cache addresses for which the true Home

node is not within the cluster. Likewise, a *Peer agent* behaves exactly like a peer node, monitoring actions that require tracking and searching for cached copies of a line in other clusters. Thus the protocol within the cluster is identical to the basic protocol, with the agents responsible for all interactions between clusters. The Peer agent and the Home agent act together, with one or the other responding to all requests in a manner indistinguishable from a local node. We will describe the Home agent and Peer agent actions presently.

Because the agents must communicate with remote clusters, delays may be substantially longer than within a cluster, both because of greater distances and because there may be additional serial messages. However, agent responsibilities have been carefully assigned so that the information needed for most common operations can be cached within the agent, allowing the agent to respond to many requests as fast as any other node in the cluster. The goal is that communications among clusters should only be necessary when there is a change in the state of the system requiring global communication regarding a cache line. For example, an intelligent agent can filter repeated invalidation requests by remembering that no copies have passed through the port since the previous invalidation request to the same line.[8]

Because the protocol is essentially identical at each level in a hierarchy, the same mechanisms can be used to handle conflicts at different levels in the hierarchy. Because of its hierarchical nature, there are no circular dependencies, and conflicts at multiple levels can be resolved serially.

2.4.1. The Home Agent

If the Home node of a cache line requested is not in the local node, the Home agent must respond. It must know the cluster containing the Home node. When a Home node request is directed to it, the agent responds by communicating in the next level of the hierarchy, using a protocol that differs only in subtle ways from the protocol previously described. It distinguishes the Home cluster from the other clusters, sending requests to all the clusters, waiting for a response, which it then forwards to the local requesting node. When it receives the followup confirm/cancel/conflict message from the requesting node, it forwards that directly to the Home cluster and awaits a response, which it again forwards to the requesting node. Note that there are actually two kinds of Home nodes corresponding to the two levels in the hierarchy. One serves as an agent for a cluster, appearing to be the Home node for all addresses with Home outside the local cluster. The other serves as an agent for incoming requests to the cluster, appearing as the Home node for all addresses with Home inside the cluster.

2.4.2. The Peer Agent

Likewise, the Peer agent responds to requests within the cluster by interacting with other clusters through essentially the same protocol, broadcast the request to agents of each of the other clusters and collecting the responses before it responds to the original requester. Peer agents also appear as two types. One serves as a surrogate for all nodes outside of the cluster that might have cached copies of the data. The other serves as a target for requests generated from other clusters relaying information about locally cache data.

The extra delay added by the hierarchical protocol can be largely overcome by the judicious use of extra data structures. By remembering the state of a cache line inside a local cluster, an agent can respond directly to a request from another cluster rather than having to broadcast the request to the local cluster. Likewise, by remembering the state of a cache line in the rest of the system, an agent can often respond to a request without having to rebroadcast the request to the other clusters. We have identified two general caches and give examples of how they can be used to reduce both latency and traffic.

In order to maintain information regarding copies of a cache line kept outside the cluster containing the Home node, we introduce a simple directory, called the Export Directory. This directory can have a variety of structures, and may be complete or partial [9]). The basic purpose is to remember information about remotely cached lines, allowing the local cluster to access line with a local Home without requiring any remote accesses. The Export Directory reduces traffic both directions. It allows the Home agent to cache the line and make it available to remote nodes without having to go to the Home node. It can also exploit the knowledge of where copies exists (or that none exist) outside the local cluster to minimize intercluster requests generated in the cluster containing the Home node.

A second structure is known as the Import cache. This structure is responsible for maintaining information about lines brought into the cache that have a remote Home node. This cache can greatly reduce intercluster traffic for common cases. For example, if the F node is in another cluster, the basic protocol requires retrieving the data from the remote node even though a local node may hold a copy of the cached line. The Import cache can act as a surrogate for the (remote) node and supply the data as if it were coming from the remote node, without having to retrieve it.

3. Performance Evaluation

To evaluate the benefits of our proposal, we compare the MESIF protocol to existing 3-hop protocols such as coherent HyperTransport™ and Scalability Port™. 3-hop protocols require the requestor to send its request directly to the home node and then letting the home node perform the snoop broadcast on its behalf[7]. The quickest response happens on a hitM or hitE where the snoop hits an M or E cached copy and the requesting node gets its line in 3-hops when the line is forwarded directly back to the requestor. HitS and hitI require "4 hops" with data from main memory. Below, the latency table compares MESIF to a 3-hop protocol.

	MESIF	3-hop
hitI/hitS	2 hops + max(mem latency, (2 hops + cache latency))	2 hops + max(mem latency, (2 hops + cache latency))
hitE/M	2 hops + cache latency	3 hops + cache latency
hitF	2 hops + cache latency	N/A

To gauge the impact of a 2-hop protocol in a point-to-point interconnect versus a 3-hop protocol, we utilized a Mean Value Analysis tool developed in-house by a performance validation group (the authors did not develop this tool). The tool is fundamentally based on the latency formulas shown in the above table.

Specifically, the tool is used to project TPC-C performance as that is the most used, and some would say, the most indicative of server performance. The tool projects performance for various MP configurations where multiple parameters such as core speed, number of threads per core, number of cores per node, various cache sizes, speed of interconnect components, memory speed and bandwidth, etc. are entered. Some of the parameters such as cache miss rates, cache miss types, and I/O activity are captured from actual systems running the OLTP workload. The algorithms employed in the tool as well as many measured and derived parameters are trade secrets and cannot be divulged in this paper. Needless to say, the tool has been validated extensively with actual small MP systems (4 sockets) and projections of absolute performance fall within +/- 5% of the actual system. When the tool is used to project relative performance, e.g. modifying one or two components of the system, the error bar is much less.

Mean Value Analysis works for applications such as OLTP because it is a throughput oriented workload where the average behavior over a long run of the application is indicative of the total behavior of the application. This also applies to applications such as SpecWeb.

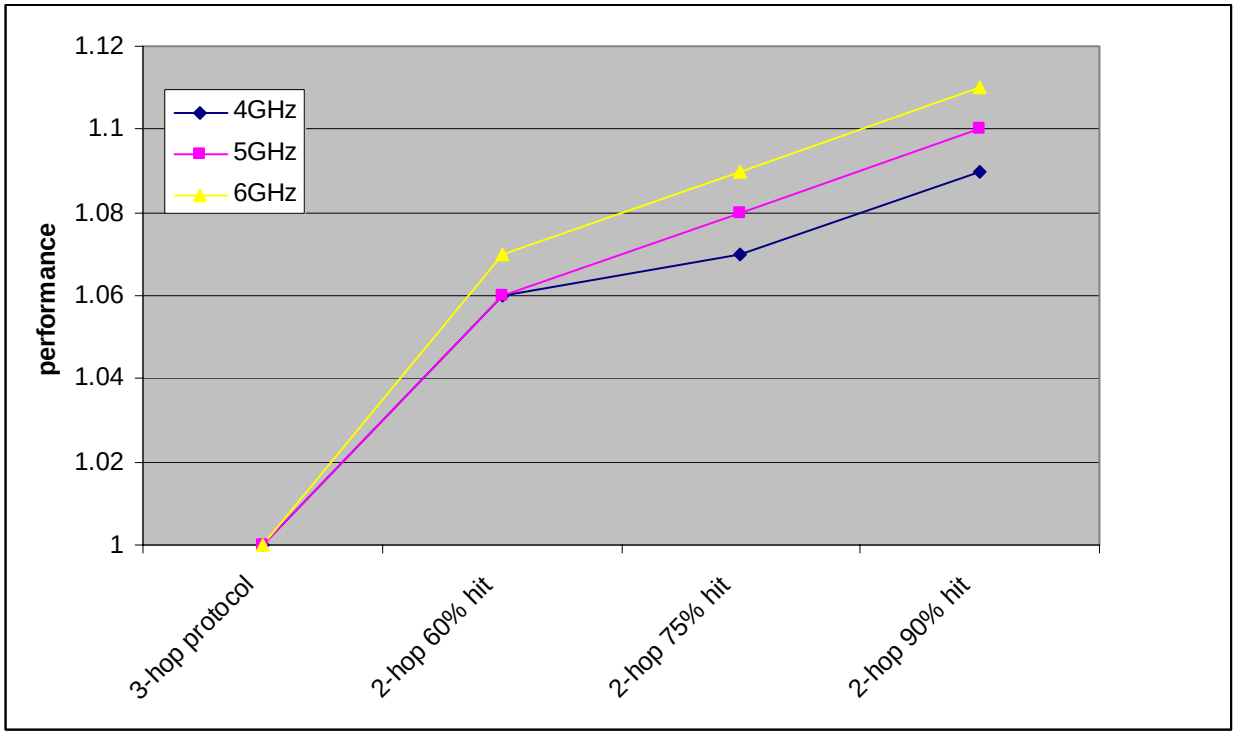
In the following table, we list the important parameters used in the projections:

Parameters	**Value**
Core frequency	4GHz, 5GHz and 6GHz
Threads per core	2
Cores per node	2
Number of sockets	4
Cache line size	64 bytes
Cache size for node	12MB, 12-way associative
Cache hit latency	~40ns (serial tag data access)
Cache miss that hit in another node's cache	60%, 75%, 90%
Aggregate memory size	32GB (8GB per socket)
Main memory latency	~100ns
Main memory latency per socket	6.4GB/s
Interconnect	Fully connected, 2 uni-directional links per port
Interconnect bandwidth	6.4GB/s per link, 12.8GB/s aggregate per port
1-hop Interconnect latency	20ns

The cache specified in the table is the largest cache in a node and is shared by the multiple cores in the node. It is listed in the table because it has the most impact on the performance of 2-hop versus 3-hop protocols. Other caches closer to the core have less impact.

In the row labeled “Cache miss that hit in another node’s cache”, we vary the percentage of requests leaving a node that can be satisfied by another node’s cache. Cache-to-cache transfers are inherently faster than requests satisfied by main memory and are another major component in determining the goodness of a 2-hop protocol.

The performance data of varying the core frequency and varying the percentage of last level cache misses that hit in a last level cache in another socket is shown below.



Our analysis indicates a clear advantage in all cases for the MESIF over the 3-hop protocol. For a 4GHz clock, with 4 nodes (four threads—two processors--per node), performance will be enhanced by 6 to 11%, depending on cache-to-cache to main memory-to-cache ratio. Obviously, as more requests can be satisfied from another node’s cache, the more benefit of a 2-hop protocol. For faster cores, the improvement is slightly larger because the impact of going off-node is larger; any improvement there will have a commensurately larger impact on performance.

4. Summary

We have described a new protocol for point-to-point interconnects that garner the latency benefits of a bus-based protocol in terms of number of hops and retains the bandwidth advantages of a point-to-point fabric. We have shown the performance improvements of our 2-hop protocol versus known point-to-point protocols that require a minimum of three hops for a memory transaction. We have also described an hierarchical extension of

the protocol that can retain the goodness of 2-hop protocols with the judicious use of caching structures: an export directory and import cache

Within our institution, this work has already spawned numerous variants of the 2-hop protocol that have been formally validated, and analyzed on performance and implementability. As we stated, this is but one part in a production-worthy interconnect and much work is required to build on top of this protocol.

5. Related Work

The hierarchical MESIF protocol is similar to the pruning cache protocol [8], which exploits the single-path property between every pair of nodes in a tree to prune away major parts of the tree, restricting broadcasts required to those portions of the tree that must participate in transactions. Like MESIF, the pruning cache assumed clusters of nodes with a unique path between every pair of clusters. Unlike MESIF, the protocol did not address serialization within a cluster, assuming that actions within the cluster could be easily ordered, either because a bus or a uni-directional ring was used.

The Token Coherence [10] protocol is contemporary, independently developed work with MESIF. Token Coherence assumes an unordered interconnect and separates the protocol into a performance protocol, which is fast but does not always succeed, and a correctness substrate, which guarantees correct ordering of operations. Common operations can frequently be retrieved in two hops, though it is not apparent how any notion of locality can be exploited, so even for small systems, the round-trip delay may be much longer than a single, point-to-point roundtrip delay.

6. References

[1] Alan Charlesworth, Alan Phelps, Ricki Williams, and Gary Gilbert. "Gigaplane-XB: Extending the Ultra Enterprise Family," *Proceedings of the Symposium on High Performance Interconnects V*, pages 97–112, August 1997.

[2] E. Hagersten, A. Landin, and S. Haridi. "DDM - A Cache-only Memory Architecture," *IEEE Computer*, September 1992, pp. 44-54.

[3] Institute of Electrical and Electronics Engineers, New York, NY. "IEEE Standard for the Scalable Coherent Interface (SCI)," August 1993. ANSI/IEEE Std. 1596-1992.

[4] S.L. Scott and J.R. Goodman, "Performance of Pruning-Cache Directories for Large-Scale Multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, **4**, No. 5 (May 1993), pp. 520-534.

[5] "HyperTransport™ Technology I/O Link," White paper, July 2001. Available at http://www.hypertransport.org/tech/tech_whitepapers.cfm

[6] Azimi, M.; Briggs, F.; Cekleov, M.; Khare, M.; Kumar, A.; Looi, L.P., "Scalability port: a coherent interface for shared memory multiprocessors," *Proc. 10th Symposium on High Performance Interconnects HOT Interconnects (HotI'02)*, pp. 65-70, August 2002.

[7] Paul Sweazey and Alan J. Smith, "A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus," *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 414–423, June 1986.

[8] S.L. Scott and J.R. Goodman, "Performance of Pruning-Cache Directories for Large-Scale Multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, **4**, No. 5 (May 1993), pp. 520-534.

[9] A. Gupta, W.-D. Weber, and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," *International Conference on Parallel Processing(ICPP)*, part I: pp. 312-321, Aug 1990.

[10] 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," *30th Annual International Symposium on Computer Architecture*, June 2003.