Delta Coherence Protocols: The Home Update Protocol*

C. Williams and P.F. Reynolds, Jr.
University of Virginia, Computer Science Department

B. R. de Supinski
Lawrence Livermore National Laboratory

Abstract

We describe a new class of directory coherence protocols called *delta coherence protocols* that use network guarantees to support a new and highly concurrent approach to maintain a consistent shared memory. Delta coherence protocols are more concurrent than other coherence protocols in that they allow processes to pipeline memory accesses without violating sequential consistency; support multiple concurrent readers and writers to the same cache block; and allow processes to access multiple shared variables atomically without invalidating the copies held by other processes or otherwise obtaining exclusive access to the referenced variables. Delta protocols include both update and invalidate protocols. In this paper we describe the simplest, most basic delta protocol, an update protocol called the *home update protocol*.

Delta protocols are based on *isotach* network guarantees. An isotach network maintains a logical time system that allows each process to predict and control the logical time at which its messages are received. Processes use isotach guarantees to control the logical time at which their requests on shared memory appear to be executed. We prove the home update protocol is correct using logical time to reason about the order in which requests are executed.

*This work was supported by NSF grant CCR-9503143, with additional funding provided under DARPA grant DABT63-95-C-0081. Portions of the work were performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48; UCRL-ID-139737.

1. Introduction

Caching data can reduce access latency and improve data availability, but in the case of writeable data, caching raises the problem of how to maintain consistency among copies. The problem appears in different guises in several different contexts: as the cache coherence problem in multiprocessors; as the problem of maintaining a distributed shared memory (DSM) in distributed computations; and as the replica control problem in distributed databases. This paper describes the *home update protocol*, a member of the class of coherence protocols called *delta coherence protocols* that uses *isotach message ordering guarantees*[1] to solve the coherence problem in a new and highly concurrent way. Our goal is to show how isotach guarantees are useful in solving the coherence problem and in reasoning about coherence protocols.

Solving the coherence problem is hard because it requires coordinating the execution order of accesses at different nodes. The traditional approach to the problem is to reduce the coordination required by limiting concurrency or weakening the correctness criteria. Existing protocols that enforce sequential consistency (SC) require that nodes execute requests one-at-a-time and invalidate or lock copies while executing write requests. Delta protocols use isotach guarantees to coordinate accesses, an approach that allows delta protocols to enforce SC without limiting concurrency. Whether delta coherence protocols outperform existing coherence protocols depends on the cost of implementing isotach guarantees and on the extent to which applications can take advantage of the high level of concurrency offered by delta protocols.

2. Isotach systems

An isotach [*Greek*: iso = same; tach = speed] system[1] implements a logical time system in which all messages *appear* to travel at the same speed — one unit of logical distance per unit of logical time. Given this property, called the *isotach invariant*, a processor can control the logical time at which each of its messages is received by controlling its logical send time.

Isotach systems use the exchange of signals called *tokens* between neighboring nodes (switches and processors) to implement a distributed logical clock. The *pulse* at a processor is the number of token waves the processor has received. An isotach system assigns a logical time to each event of sending or receiving a message. An isotach logical time is a lexicographically ordered 3-tuple in which the first and most significant component is the *pulse* at the processor at which the send or receive event occurs. The remaining two components, the *pId* and *rank*, are tie-breakers used to order events that occur in the same pulse. Events with the same pulse component are ordered by the pId of the sender. Events with the same pulse and pId components are ordered by the rank, i.e., issue order, of the message.

The isotach logical time system extends Lamport's logical time system[2] by guaranteeing that send and receive times are consistent with the isotach invariant: each message travels one unit of logical distance per pulse of logical time. Isotach systems can implement a variety of distance metrics[3]. Here, dist(p,p') , the logical distance from node *p* to node *p*' , is the routing distance from *p* to *p*' , i.e., the number of switches traversed by a message sent by *p* to *p*' . For any message *m* sent by *p* to *p*' , *d(m)*, the logical distance message *m* travels, is dist(p,p'). For simplicity, we assume distances are static. Distances may be asymmetric, i.e. dist(p,p') does not necessarily equal dist(p' ,p). By the isotach invariant, for any message *m*, *m*'s logical receive time is exactly d(m) pulses after *m*'s logical send time, i.e., $t_r(m) = t_s(m) + d(m)$. (The scalar quantity d(m) is added to the tuple $t_s(m)$ by adding d(m) to the pulse component of the tuple.) Assuming each processor executes messages in receive order, a message's logical receive time can be used as its logical execution time. Thus, for any message *m*, $t_x(m)$, the logical execution time of *m*, equals $t_r(m)$. This assumption is for simplicity and is stronger than necessary. Execution times can be shifted in relation to receive times in any way that preserves receive order. Furthermore, in an execution in which messages are operations on shared variables, operations can be executed in any order that preserves the receive order among *conflicting* operations. Two operations conflict if they access the same variable and are not both reads.

Most delta protocols require an isotach system that supports *predictable responses*. A predictable response is a message *m'* sent in response to another message *m* such that the send time of *m'* can be predicted from the receive time of *m*, i.e., $t_s(m') = t_r(m) + c$. For simplicity, we assume *c* is 0. (In any practical system, *c* is a small tunable system constant, greater than zero.) Given the isotach invariant and knowledge of the logical distances involved, the receive time of *m'* can be predicted from the send time of *m*: $t_r(m') = t_s(m) + d(m) + d(m')$. A predictable response inherits the pId and rank components of the original message.

Each processor has a switch interface unit (SIU) that tracks logical time and acts as the interface between applications and the isotach system. An application can simply assume that its messages will appear to be executed in the order issued. Given the isotach invariant and the assumption that messages are executed in receive order, an SIU can control the relative order in which locally issued messages appear to be executed. In particular, an SIU can ensure that a batch of messages appear to be executed at the same time by sending the messages so that they are received in the same logical pulse and can ensure that messages issued in a sequence appear to be executed in sequence by sending the messages so that they are received in non-decreasing pulses.

Isotach systems can be implemented using the *isonet* algorithm[1], in which network switches route messages in logical time order. Alternatively, the work of ordering messages can be shifted to the SIUs to permit the use of commodity switches. A prototype system based on this approach has been implemented on a cluster of commodity PCs connected with Myrinet[4]. Both algorithms are scalable, requiring the exchange of tokens only among nearest neighbors. In the prototype, which implements isotach functionality in software, the round-trip user-to-user level latency of isotach messages is on the order of 50 usec, about twice that of non-isotach messages on the same hardware[5]. To further reduce the cost of maintaining isotach guarantees, we are re-designing the messaging layer software and building a second generation prototype with custom SIUs.

3. Model

The coherence problem occurs in several contexts, each with its own terminology. The terms used here are from the literature on the cache coherence problem in multiprocessors. We rely on the reader interested in DSM or replica control to make the appropriate translations.

We consider a system consisting of multiple processors connected to a memory system. Processors issue read and write *requests* to the memory system. A write request (WRITE) on variable *v* instructs the memory system to assign a specified value to *v*; a read request (READ) on *v* instructs the memory system to return the value of *v*. A variable is *shared* if more than one processor can issue requests on it. We consider only shared variables.

The memory system encapsulates the representation of (shared) memory and the procedures for accessing it. The processor/memory system interface is as follows:

- processors issue READs and WRITEs to the memory system. (To enable the processors to specify the variable to be accessed, the processors and memory system share a naming scheme for variables.)
- the memory system returns a value in response to each READ.

The internal details of the memory system are not visible to the processors.

A memory system consists of interconnected memories and controllers programmed to execute the system's coherence protocol. The memory space is partitioned across the memory modules (MM). Each processor has a cache memory and cache/coherence controller (CC), which handles locally issued memory requests and manages the cache. In a delta protocol, the CC also performs the functions of the SIU, i.e., it tracks logical time and controls the logical time at which local operations are sent.

For each variable *v*, the primary copy, called the *home copy*, is located in an MM. The MM containing

v's home copy is *v*'s *home.* Secondary copies, called *cache copies,* may be stored in the cache memories. The number of cache copies of *v* can vary dynamically from zero to the number of processors in the system. A request for *v* is a *hit* if a copy of *v* is in the issuing processor's cache; otherwise it is a *miss*.

The memory system translates requests into *operations,* executes the operations, and returns a value for each READ. An operation reads, writes, creates, or destroys a copy of a variable. For each locally issued WRITE, the CC generates one or more write operations (writes) and for each locally issued READ, the CC generates a single read operation (read). The phrase "the execution of request *R* on copy *c*" means the execution of the operation resulting from *R* that is executed on copy *c*.

In a delta protocol, each operation is sent as an isotach message. The logical distance, send, receive, and execution times of an operation are the logical distance, send, receive, and execution times of the message carrying the operation. An operation on the local cache copy is sent as a self-message. A self-message is an isotach message sent by a processor to itself. Since self-messages do not enter the network, for any self-message *m*, d(m) = 0 and $t_r(m) = t_s(m)$. Fig. 1 summarizes terms relevant to operations in a delta protocol.

for copy *c*		
$\delta(c)$	The delta of *c*. In home update protocol, dist(home,c)	
for operation op executed on copy *c*		
$t_s(op)$	send time of op	} related by isotach invariant: $t_r(op)=t_s(op)+d(op)$
$t_r(op)$	receive time of op	
$d(op)$	logical distance traveled by op	
$t_x(op)$	execution time of op	$t_x(op)=t_r(op)$, by assumption
$t_{effx}(op)$	effective execution time of op	$t_{effx}(op)=t_r(op)-\delta(c)$
xdist(op)	execution distance of op	$xdist(op)=t_{effx}(op)-t_s(op)=d(op)-\delta(c)$

Fig. 1. Delta coherence protocol terms and notation.

Each copy in a delta protocol is assigned a *delta*. In the home update protocol, the delta of copy *c*, denoted $\delta(c)$, is dist(home,c). (Thus, the delta of a home copy is zero.) A copy's delta represents the number of logical time pulses by which the copy lags behind the home copy. For any operation *op* on copy c, the *effective* execution time of *op*, denoted $t_{effx}(op)$, is $t_x(op) - \delta(c)$. Informally, $t_{effx}(op)$ is the logical time at which *op appears* to execute, i.e., *op*'s logical execution time adjusted to compensate for *c's* delta. For any operation *op*, *xdist(op)*, the *execution distance* of *op* is defined as $t_{effx}(op) - t_s(op)$. Thus, $xdist(op) = d(op) - \delta(c)$, where *c* is the copy on which *op* is executed.

4. Correctness criteria

The most basic task of a coherence protocol is to make replication transparent to the processors. The result of any execution should be as if the requests of the processors were executed on a single-copy memory, i.e., a memory containing a single copy of each variable. Coherence protocols may enforce the following ordering properties:

- SC. A memory system enforces SC if "[t]he result of any execution is as if the [requests] of all the processors were executed in some sequential order, and the [requests] of each individual processor appear in this sequence in the order specified by the program"[6]. The execution shown in Fig. 2 violates SC since no sequential ordering of the requests can produce the results shown.

P1::
WRITE(v)1;
WRITE(w)1
P2::
READ(w)1;
READ(v) 0

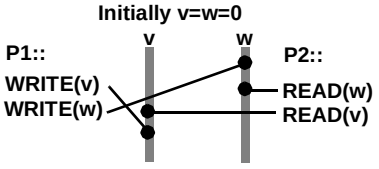

Fig. 2. Violating SC.

- Atomicity. The memory system should execute requests issued as part of the same transaction or atomic

action atomically, i.e., so that the requests appear to be executed as an indivisible unit. Thus, the result of any execution should be as if the requests of all the processors were executed in some sequential order and the requests in each transaction appear in this sequence as a contiguous subsequence, not interleaved with requests from other transactions.

Here, we use the term *atomicity* to mean consistency atomicity, not failure atomicity, i.e. the guarantee is about the relative order in which requests appear to be executed, not about the results of a failure. Coherence protocols may also be required to enforce *failure atomicity*: the operations resulting from each request (and the operations resulting from all requests in the same transaction) should be executed on an all-or-nothing basis. Failure atomicity is an important concern in distributed databases, but a fault-free system is normally assumed in a the context of multiprocessor cache coherence and is often left to separate mechanisms in the DSM context. The isotach prototype uses a sender-based protocol and a reliable network (Myrinet) to achieve reliable communication. An unreliable network would require use of a commit protocol.

The relative importance of SC and atomicity depends on the context. With a few exceptions, cache coherence protocols for multiprocessors and DSM protocols focus on SC (or a weaker variant) and leave the task of enforcing atomicity to separate mechanisms. On the other hand, databases focus on enforcing atomicity. The high cost of enforcing SC and atomicity has led to extensive exploration of weaker memory consistency models. Whether the resulting improvement in performance justifies the more complex memory interface is an undecided issue[7]. Delta protocols enforce atomicity and SC using isotach ordering guarantees without the locks and restrictions on pipelining required in conventional systems. Thus, delta protocols represent an alternative to weakening the guarantees offered by the memory system.

5. Home update delta coherence protocol

The home update protocol is the simplest of the delta protocols and serves as the basis for the other delta protocols [3], which include invalidate as well as update protocols. As indicated by the name, the home update protocol is an update protocol in which the home is responsible for distributing updates.

State Information

The protocol is a directory protocol. The home for each variable *v* stores a *directory* recording the set of processors with cache copies of *v*. For simplicity, we assume a bit vector representation for the directory: bit *i* in the bit vector for *v* is set *iff* processor *i* has a cache copy of *v*. (Any of several proposals for improving the scalability of directories, e.g., the method used in the Alewife[8] machine, could be used instead.)

Each CC stores a bit with each line in the local cache that indicates whether the line is currently allocated. When a CC schedules a miss request on *v*, it creates a a local copy of *v* by marking a currently unallocated cache line allocated. The CC destroys a copy by sending a release and marking the cache line unallocated (see Executing Requests, below). For each allocated cache line, the CC stores the name of the variable to the which cache line is allocated and an *outstanding request count*. A CC releases a cache copy of *v* only if the outstanding request count for *v* is zero. An *outstanding request* is a locally issued request that has been scheduled but not *completed*. A READ hit completes when the read is executed; a READ miss when the read response is executed; and a WRITE when the own-update is executed (see Executing Requests, below). The outstanding request count can be represented using two bits per variable if each processor is limited to one outstanding read and one outstanding write per variable[9]. Alternatively, since we expect a processor to have only a few outstanding requests at any given time, the counts can be maintained at a high granularity, e.g. on a page basis.

Except for requiring that the CC not destroy any cache copy to which it has outstanding requests, we do not specify the replacement policy. Specific replacement policies may require additional state information. The state information required to support a competitive policy, a strategy that destroys copies that are infre-

quently referenced but often updated[10], may subsume the outstanding request count.

Executing requests

When processor *p* issues a request, its CC translates the request into one or more operations, called *initiating operations.* In the home update protocol, each request results in exactly one initiating operation. A READ or WRITE miss issued by *p* results in the creation of a new cache copy at *p*. When a CC schedules a miss, the CC creates a new cache copy by allocating a cache line. (The scheduling algorithm ensures that any subsequent access to the cache line can occur only after the copy has been initialized.) If all lines are allocated, the CC first destroys a local cache copy. A CC destroys its copy of *v* by sending a *release* message to *v*'s home. The home executes a release on *v* by removing *p* from *v*'s directory. When the CC sends the release, it marks the cache line unallocated.

Other actions taken by the memory system in executing a request depend on the request type.

READ miss

The CC generates a read on the accessed variable *v*'s home copy and schedules the sending of the read (described below). On receiving the read, *v*'s home adds *p* to the directory for *v* and sends a read response message to *p*. The CC executes a read response by assigning the value returned to the (new) cache copy of *v* and returning the value to *p*.

READ hit

The CC generates a read on the cache copy of *v* and schedules the sending of the read. (Recall that an operation on the local copy is sent as a self-message and has a logical send and receive time even though it does not enter the network.) At the logical receive time, the CC executes the read on its cache copy, returning the value to *p*.

WRITE (hit or miss)

The CC generates a write on *v*'s home copy and schedules the sending of the write. The write is sent to *v*'s home even if *p* has a cache copy of *v.* On receiving the write, *v*'s home assigns the value to the home copy, adds *p* to the directory for *v,* if *p* is not already in the directory, and sends a write to every processor in *v*'s directory (including *p*). Writes sent by the home are usually called *updates.* An *own-update* is an update received by a CC in response to its own write. A CC executes an update on *v* by assigning the value transmitted by the update to its cache copy of *v.* If the CC has no cache copy of *v,* it discards the update. This case can occur if the CC recently released *v.*

Using isotach guarantees

The home update protocol, as described so far, is similar to other update protocols. The home update protocol differs from other update protocols in its use of isotach guarantees:

- The isotach invariant allows the sender of each operation to control its logical receive time.
- Sending updates and read responses as predictable responses allows the definition of copy deltas, establishing the relationship between the cache and home copies of each variable.

Given the ability to control logical receive times and to relate cache copies to home copies, CCs can control the effective execution time of requests by controlling the send times of the initiating operations.

To enforce sequential consistency, each CC applies the following rule in scheduling each request: *send the initiating operation so that its effective execution pulse is no less than that of the previous request*. The *effective execution pulse* is the pulse component of the effective execution time. A request with the same effective execution pulse as the previous request will have a later effective execution *time* due to its rank component.

```
lastR = 0
for each request R issued by p
  if R is a READ hit
    xdist = dist(home,p)
  else
    xdist = dist(p,home)
  sendp = max(lastR-xdist,now)
  lastR = sendp;
```

Fig. 3. Scheduling algorithm.

As shown in Fig. 3, the CC remembers *lastR*, the effective execution pulse of the last request it scheduled, and sends the initiating operation for each new request so that its effective execution time is no less than *lastR*. If the current logical time (*now*) plus the execution distance of the initiating operation is less than *lastR*, the CC sends the operation at the earliest time at which it can be sent and have an effective execution pulse no less than *lastR*. Otherwise, the CC sends the operation immediately.

When *v*'s home executes a write on *v*, it sends updates to the processors in *v*'s directory as predictable responses to the write. Sending updates as predicatable responses allows the assignment of deltas to cache copies. Each cache copy *c* is updated exactly $\delta(c)$ after the home copy. As will be shown below, sending updates as predictable responses also ensures that all writes (the initiating write and any updates) resulting from the same WRITE have the same effective execution time, with the result that each WRITE appears to execute atomically. Although the *execution* times of the operations may differ, the *effective execution* times are the same because the delta of each copy exactly compensates for the time required to propagate the value from the home.

Similarly, when *v*'s home executes a read from *p*, it sends the read response to *p* as a predictable response. Sending the read response as a predictable response means that the new copy created at *p* will be initialized with a "timely" value, i.e., the value of the home copy $\delta(c)$ pulses before. We will show that sending the read response as a predictable response also means that the intialization of the new local copy has the same effective execution time as the initiating operation resulting from the READ miss. Since any subsequent access to the local copy will have a later effective execution time, the new copy will not be accessed until after it is initialized.

Timing Diagram

Fig. 4 shows the relationship between the logical send, receive, and effective execution times for each operation generated by a READ hit, a READ miss, and a WRITE issued at processor *p*. The effective execution time for each operation shown is the same, time *t*. Processors *p'* and *p"* represent other processors in the directory for *v*.

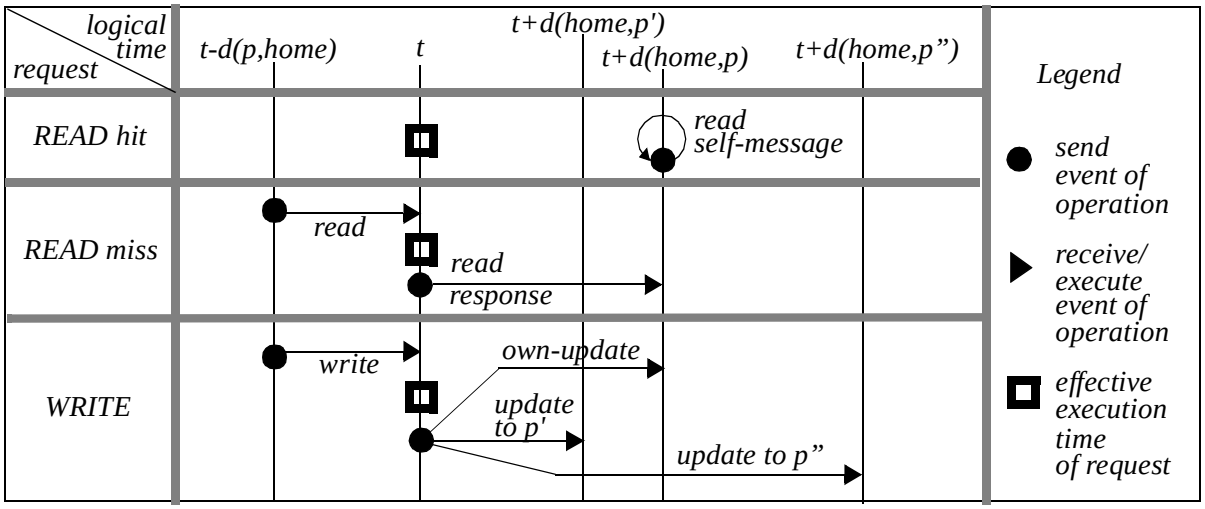

Fig. 4. Timing diagram for operations

As shown in the figure, to achieve an effective execution time of *t*, the read operation resulting from a READ hit at processor *p* should be executed at time *t* +dist(home,p). Since the read resulting from a READ hit is sent as a self-message, the read is sent and received at this same time, *t*+dist(home,p). The initiating operation resulting from a READ miss or a WRITE should be sent at *t*-dist(p,home) to achieve an effective execution time of *t*. Since an initiating operation resulting from a READ miss or a WRITE is executed on the home copy, its execution time and effective execution times are the same. Any update or read response sent by the home will have the same effective execution time as the initiating operation since the delta of the copy on which the update is executed compensates for the propagation delay.

Under the home update protocol, a node can execute any number of requests concurrently, i.e., if the processor *p* submits *r* requests, the local CC can schedule all *r* requests so that they have the same effective execution pulse and complete within d(p,home) + d(home,p) logical pulses. By contrast, under a typical directory protocol[11], the *r* requests must be handled serially, i.e., request *i* cannot be sent until after request *i*-1 completes.

6. Proof of Correctness

We prove the correctness of the protocol. We show that memory system *M* is correct by showing that the result of any execution on *M* is the same as if it were executed on a memory system *M*' known to be correct. Showing that a correct memory system can be substituted for *M* without changing the result of any execution implies that *M* is correct. In particular, we show that for any execution *E* of any program *P* on a memory system that uses the home update protocol, there is an equivalent sequential execution *S* of *P* on M' , where *S* is SC. Memory system *M*' executes requests serially in some sequential order on a single-copy memory, translating each request into a single operation. From the viewpoint of the memory system, a *program* is a sequence of requests, in the order in which they are submitted, and an *execution* of a program *P* is the sequence of operations resulting from *P* in the order in which they are executed. Executions *S* and *E* of *P* are *equivalent* if every READ in *S* returns the same value as the corresponding READ in *E*.

DEFINITION. For any request *R*, the *effective execution time of R* is the effective execution time of the initiating operation resulting from *R*.

LEMMA 1. *The effective execution times of requests derived from execution E of program P define a total order over the requests in P.*

Proof. Since each logical time is a 3-tuple in which the second and third components serve as tie-breakers, each initiating operation for a request in *E* has a unique effective execution time. □

DEFINITION. For any program *P*, Let *P*' be the permutation of *P* in which the requests in *P* appear in increasing order by their effective execution times.

DEFINITION. Let *S* be the execution of *P* in which the requests in *P* are executed on M' in the order in which the requests appear in *P*' .

LEMMA 2. *All operations resulting from the same request have the same effective execution time.*

PROOF. Since a READ hit results in only one operation, the claim is trivially true for READ hits. A READ miss at processor *p* results in an initiating read operation *r* executed on the home copy and a read response *r*' sent by the home to *p*. Since the delta of a home copy is 0, $t_{effx}(r) = t_r(r)$. Since the home sends *r*' as a predictable response to *r*, $t_{effx}(r') = t_r(r) + dist(home,p) - \delta(c)$, where *c* is the copy created at *p* as a result of the miss. Since $\delta(c) = dist(home,p)$, $t_{effx}(r') = t_r(r) = t_{effx}(r)$.

The WRITE case is similar to the READ miss. A WRITE results in an initiating write operation *w* executed on the home copy and updates sent as predictable responses to each processor with a cache copy. Since the delta of a home copy is 0, $t_{effx}(w) = t_r(w)$. For any update *u* sent by the home to processor *p*'

with copy *c'* of *v* in response to *w*, since the home sends *u* as a predictable response to *w*, $t_{effx}(u) = t_r(w) + dist(home,p') + \delta(c') = t_{effx}(w)$. Since $\delta(c') = dist(home,p')$, $t_{effx}(u) = t_r(w) = t_{effx}(w)$. □

DEFINITION. For any READ *R* on any variable *v* (or for any read operation *r* resulting from *R*), the *logically preceding WRITE* is the WRITE on *v* with the greatest effective execution time that is less than the effective execution time of request *R* (or operation *r*).

We assume each variable is written before it is read. Thus every READ has a logically preceding WRITE. To model programs that read uninitialized variables, a sequence of WRITEs, one for each variable, can be prepended to each program, where each WRITE assigns an arbitrary value.

DEFINITION. Copy *c* of variable *v* is *valid* at logical time *t* if a read *r* executed on *c* at *t* returns the value assigned by the logically preceding WRITE for *r*.

Since requests are executed serially in *S* in order of their effective execution times, every READ in *S* returns the value of the logically preceding WRITE. We show the equivalence of *E* and *S* by showing that every READ in *E* is executed on a valid copy and therefore every READ returns the same value in *E* and *S*, the value of the logically preceding WRITE.

LEMMA 3. *Every read to a home copy in E is to a valid copy.*

PROOF. Let *r* be a read executed on home copy *c* at time *t*. Since $t_{effx}(r) = t$, the logically preceding WRITE for *r* is the WRITE with the greatest effective execution time that is less than *t*. Since *c* is the home copy of *v*, all WRITEs on *v* in *P* are executed on *c*. Since δ(c) is zero, each WRITE is executed on *c* at its effective execution time. Thus, the value of *c* at *t* is the value assigned by the logically preceding write. □

The reader may wish to refer to Fig. 5 while reading Lemma 4. The diagram in part a) shows the first case discussed in the proof of Lemma 4 and the diagram in part b) shows the second case. In each diagram, the top line represents events at the home copy of variable *v* and the bottom line, events at copy *c* at processor *p*. In each diagram, the thick gray line shows the transmission of the value of the logically preceding WRITE to *c*.

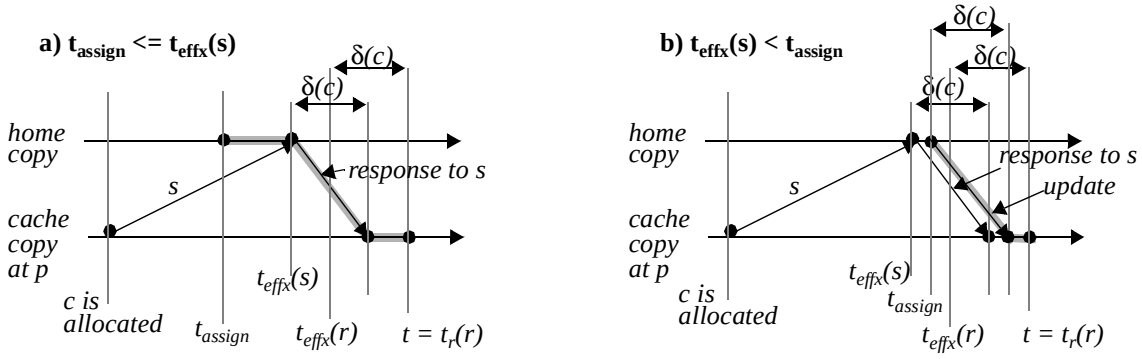

Fig. 5. Validity of cache copy *c* at time *t*.

LEMMA 4. *Every read to a cache copy in E is to a valid copy.*

PROOF. Let *r* be a read executed on cache copy *c* at processor *p* at time *t*. The logically preceding WRITE for *r* is the WRITE with the greatest effective execution time that is less than $t_{effx}(r) = t - \delta(c)$. Let *W* be the logically preceding WRITE for *r*, *V* be the value written by *W*, and t_{assign} be the time *V* is assigned to the home copy of *v*. Since the effective execution time and execution time are the same for an operation executed on a home copy, t_{assign} is also the effective execution time of *W*, and is, thus, less than $t_{effx}(r)$. We show that *V* is the value of *c* at time *t* by showing that *p* receives *V* before *t* and does not discard or overwrite the value before *t*.

Since *r* is executed on cache copy *c*, *c* was already allocated when *r* was scheduled. Let *s* be the initia-

ting operation for the READ or WRITE that causes *c* to be allocated. The home for *v* executes *s* at $t_r(s) = t_{effx}(s)$. Since the request that results in *s* is issued before the request that results in *r*, by the scheduling algorithm (Fig. 3), $t_{effx}(s) < t_{effx}(r)$. There are two cases to consider:

1) If $t_{assign} <= t_{effx}(s)$, then *W* is the last WRITE executed on the home copy of *v* through $t_{effx}(s)$. By definition of *W*, there is no WRITE with an effective execution time in the interval from t_{assign} to $t_{effx}(r)$. Since $t_{effx}(s) < t_{effx}(r)$, there is no WRITE with an effective execution time in the subinterval t_{assign} to $t_{effx}(s)$. Thus, $c = V$ at $t_{effx}(s)$ and *p* receives *V* as a response to *s* at $t_{effx}(s) + \delta(c)$. Since $t_{effx}(r) = t - \delta(c)$, $t_{effx}(s) + \delta(c) < t$. Thus, *p* receives *V* before *t*.

2) If $t_{effx}(s) < t_{assign}$, the home for *v* sends *V* to *p* as an update operation at time t_{assign}. The home adds *p* to the directory for *v* at $t_{effx}(s)$. Since *c* is allocated when *r* is scheduled, *p* does not send a release between $t_s(s)$ and the time it schedules *r*. After *p* schedules *r*, *p* cannot send a release on *v* until after *r* completes at time *t*. Thus, *p* remains in the directory through time $t + dist(p,home)$. Since $t_{assign} < t$, *p* is in the directory for *v* at t_{assign}. Thus, *p* receives *V* as an update operation at $t_{assign} + \delta(c)$. Since $t_{assign} < t_{effx}(r) = t - \delta(c)$, $t_{assign} + \delta(c) < t$. Thus, *p* receives *V* before *t*.

Since *c* is allocated at time $t_s(s)$, before *p* receives *V*, and is not destroyed until after *r* completes at time *t*, *p* assigns *V* to *c* and does not destroy *c* before time *t*.

We show by contradiction that *W* is the last WRITE executed on *c* through *t*. Let *W'* be a WRITE executed on *c* after *W* and no later than *t*. Both *W* and *W'* have an effective execution time $\delta(c)$ pulses before they are executed on *c*. Thus the effective execution time of *W'* intervenes between the effective execution times of *W* and *r*, contradicting the assumption that *W* is the logically preceding WRITE for *r*. □

LEMMA 5. *Every READ in P returns the same value in S and E.*

PROOF. Consider read *r* resulting from READ *R*, where *r* is executed on copy *c* in *E*. Since every copy is either a home copy or a cache copy, by Lemmas 3 and 4, *c* is valid when *r* is executed. By Lemma 2, *r* and *R* have the same effective execution time and the same logically preceding WRITE. Thus *R* returns the value of the its logically preceding WRITE in *E*. Since requests are executed in *S* serially, on a single-copy memory, in order by their effective execution times in *E*, *R* returns the same value in *S* and *E*. □

LEMMA 6. *Execution S is SC.*

PROOF. Consider any two requests *R* and *R'* issued by the same processor *p*, where *R* is issued before *R'*. Let *op* be the initiating operation for *R* and *op'* be the initiating operation for *R'*. By the scheduling algorithm (Fig 3), the CC chooses $t_s(op')$ such that $t_{effx}(op') > t_{effx}(op)$. Thus, *R'* appears after *R* in *P'* and is executed after *R* in *S*. □

THEOREM. *The protocol is correct.*

PROOF. By Lemma 5, *E* and *S* are equivalent. By Lemma 6, *S* is SC. Thus, the result of any execution on a memory system using the home update protocol is the same as if it were executed on a single-copy memory in some sequential order consistent with the program order. □

7. Atomicity

An *isochron* is a group of requests issued as a batch and executed atomically. The home update protocol can be adapted to execute isochrons atomically by substituting the scheduling algorithm in Fig. 6 for the algorithm in Fig. 3. In the revised algorithm, the CC schedules requests at the isochron granularity, i.e., it schedules requests so that all requests in the same isochron have the same effective execution pulse. The CC continues to enforce SC by scheduling each isochron so that it has an effective execution pulse no less

than the previously scheduled isochron.

We show that the requests in each isochron are executed atomically by showing that the requests occur in the equivalent serial execution *S* as a contiguous subsequence. Since all requests in the same isochron have the same effective execution pulse and are issued by the same processor as a batch, no other request can have an intervening effective execution time. Thus all requests in the same isochron are executed in *S* atomically. Since *E* and *S* are equivalent, isochrons are also executed atomically in *E*.

Since the requests in an isochron must be issued as a batch, isochrons cannot contain internal data dependences. However, atomic actions with internal data dependences can be implemented using isochrons together with a class of operations called *split operations*[9].

```
lastiso = 0          lastiso tracks the effective execution pulse of the last isochron
for each isochron issued by the processor p
  isodist = 0;       at end of next loop, isodist = max xdist over all requests in isochron
  for each request R in current isochron
    if R is a READ hit  compute xdist for initiating op
      xdist = dist(home,p)
    else
      xdist = dist(p,home);
    remember the xdist computed for each request in the isochron; the value is used below to determine send time
    isodist = max(isodist,xdist); isodist is max execution distance over requests in isochron
  lastiso = max(now + isodist, lastiso); choose effective execution time for isochron >= lastiso
  for each request R in isochron schedule the initiating op for each request in the current
    sendpulse = lastiso - xdist of R; isochron so that t_effx(op) = isoeffx
```

Fig. 6. Scheduling algorithm extended to enforce atomicity.

8. Conclusions

In delta protocols, each copy *c* has a *delta*, δ(c), equal to the number of logical pulses by which the copy lags behind the home copy. The deltas allow nodes to control the order in which requests appear to execute and facilitate proving delta protocols correct.

Delta coherence protocols use isotach guarantees to enforce SC with fewer restrictions on concurrency than existing protocols:

- Requests can be pipelined. Existing protocols that enforce SC require that the execution of a request not start until the execution of the previous request issued by the same processor completes[12]. (Adve and Hill have proposed an SC protocol that allows nodes to overlap the execution of a WRITE with another request, with a restriction that the effect of the second request cannot be visible to any node until after the WRITE is globally performed[13].) Delta protocols can overlap the execution of requests, requiring only that a request not appear to complete before the previous request completes, i.e., that its effective execution time not precede that of the previous request.
- No acknowledgements are required. Existing protocols use acknowledgements to inform a node when its WRITE completes. Reliance on acknowledgements adds message traffic and, more importantly, increases latency — a node delays executing a request not just until the completion of the previous request, but until it receives acknowledgement of the completion. In delta protocols, a node determines from local information the completion time of each request before it sends the initiating operation.
- Multiple processors can write the same variable concurrently. Invalidate protocols do not permit concurrent writes, though traditional update protocols do, subject to the restriction that writes are not

immediately readable.

- Writes are immediately readable. In the absence of strong message ordering guarantees, existing protocols that ensure SC cannot return the value of a read to a cache copy until the WRITE that supplied that value is globally performed (i.e., until all cache copies are updated or invalidated)[12]. This requirement is easy to satisfy in invalidation protocols, but hard in update protocols.
- Processors can execute multiple requests atomically without locks. Most existing protocols that enforce atomicity use two phase locking. Alternatively, transactions can be timestamped and restarted if they cannot be executed in timestamp order. Delta protocols allow a processor to access multiple variables atomically without locks or restarts. Processors can execute isochrons without synchronizing or obtaining exclusive access to the variables accessed.

Delta protocols offer a significantly higher level of concurrency than existing coherence protocols, while a prototype isotach network implementation demonstrates that the cost of providing this additional concurrency is low.

References

1. P.F. Reynolds, Jr., C. Williams, and R.R. Wagner, Jr., “Isotach networks,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 4, pp. 337-348, April 1997.
2. L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558-565, July 1978.
3. B. R. de Supinski, “Logical time coherence maintenance,” Ph.D. thesis, University of Virginia, 1998.
4. N.J. Boden et al., “Myrinet: a gigabit-per-second local-area network,” IEEE Micro, Vol 15, No. 1, Feb 1995, pp. 29-36.
5. J. Regehr, “An isotach implementation for Myrinet,” Technical Report CS-97-12, Dept. of Computer Science, University of Virginia, May 1997.
6. L. Lamport, “How to make a multiprocessor computer that correctly executes multiprocessor programs,” *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690-691, Sept. 1979.
7. M.D. Hill, “Multiprocessors should support simple memory-consistency models,” *Computer*, vol. 31, no. 8, pp. 28-34, August 1998.
8. D. Chaiken, J. Kubiatowicz, and A. Agarwal, “LimitLeSS directories: a scalable cache coherence scheme,” *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 224-234, April 1991.
9. C. Williams, “Concurrency control in asynchronous computations,” Ph.D. thesis, University of Virginia, 1993.
10. Karlin, A.R. , M.S. Manasse, L. Rudolph and DD. Sleator, “Competitive snoopy caching,” *Algorithmica*, Vol. 3, No. 1, pp. 79-119, 1988.
11. Censier, L.M. and P. Feautrier, P, “A new solution to coherence problems in multicache systems,” *IEEE Transactions on Computers*, 27(12):1112-1118, December 1978.
12. S.V. Adve and K. Gharachorloo, “Shared memory consistency models: a tutorial,” Computer, vol. 29, no. 12, pp. 66-76, December, 1996.
13. S.V. Adve and M. Hill, “Weak ordering — a new definition”, Proceedings of the 17th International Symposium on Computer Architecture, May 1990.
14. A.E. Condon, et al., “Using Lamport clocks to reason about relaxed memory models,” In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, Orlando, Florida, January 1999.
15. A. Landin, E. Hagersten, and S. Haridi, “Race-free interconnection networks and multiprocessor consistency,” *Proceedings of 18th International Symposium on Computer Architecture*, pp. 106-115, 1991.
16. E.E. Bilir, R.M. Dickson, Y. Hu, M. Plakal, D.J. Sorin, M.D. Hill, and D.A. Wood, “Multicast snooping: a new coherence method using a multicast address network,” In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, Georgia, May 1999.
17. A. Fekete, et al., “Implementing sequentially consistency shared objects using broadcast and point-to-pont communication,” *Journal of the ACM*, Vol. 45, No. 1, January 1998, pp. 35-69.