DELTA-CACHE PROTOCOLS: A NEW CLASS OF CACHE COHERENCE PROTOCOLS

Craig Williams
Paul F. Reynolds, Jr.

Computer Science Report No. TR-90-34
December, 1990

Delta-Cache Protocols: A New Class of Cache Coherence Protocols*

Craig Williams
Paul F. Reynolds, Jr.

Abstract: We describe a family of cache coherence protocols for MIN-based multiprocessors. These protocols, called *delta−cache* protocols, are more highly concurrent than other directory protocols. They allow more operations to be pipelined, support multiple readers and writers to the same cache block, and allow processes to execute atomic actions on multiple shared variables without acquiring exclusive access rights to the variables. The protocols are based on the *isotach* network, a network implementing a logical time system in which all operations travel at the same velocity, one switch per logical time pulse. Isotach networks are feasible. They can be implemented by applying a standard list-merge algorithm to the operations arriving on the inputs to each switch. We prove the correctness of delta-cache protocols with a new correctness criterion that integrates cache coherence with other aspects of concurrency control. We also describe a highly concurrent migration algorithm based on the isotach network.

1. INTRODUCTION

The cache coherence problem arises in multiprocessors with private caches. Caches serve to reduce memory latency and load on the interconnection network (ICN) and memory modules (MM's) by enabling processes to access local copies of shared variables, but they also introduce a problem: keeping the cache copies consistent with main memory and with each other. This paper proposes a new approach to the cache coherence problem based on locally synchronous routing within the ICN and describes a family of cache coherence protocols called *delta−cache* protocols that use this approach.

The delta-cache protocols are hardware directory protocols. Hardware protocols manage caches dynamically without direction from the programmer. They require run-time communication to maintain memory coherence, but, for this reason, are less conservative than software protocols [ChV88, MiB89], protocols that depend on the programmer or compiler to manage caches. Hardware protocols are typically classified as either snoopy or directory protocols. Snoopy protocols, designed for bus-based systems, broadcast cache updates. Directory protocols [Aga88, CeF78], intended for multiprocessors that do

The work of Craig Williams was supported by an assistantship in parallel processing sponsored by DARPA and NASA and administered by the University of Maryland.

not communicate over a bus, record the location of cached copies and send updates only to the affected caches. Directory protocols are more scalable than snoopy protocols, but existing directory protocols lose some concurrency realizable by snoopy protocols. Directory protocols maintain memory coherence by allowing at most one writer per block at a time. Multiple processors can hold read-only copies of the same block, but a write invalidates all other copies of the block. The need for writers to acquire exclusive permission increases the execution time for both readers and writers. Interleaved accesses by different processors to the same cache block can result in the cache block thrashing among the processors. Some snoopy protocols, by contrast, allow multiple readers and writers to the same cache block. The DEC Firefly and Xerox Dragon [ArB86] use the serialization of concurrently issued writes imposed by bus contention to ensure updates to the same block are received in a consistent order at all processors with copies. Though these snoopy protocols have the advantage of allowing multiple readers and writers their reliance on broadcasting all updates severely limits scalability.

Most of the delta-cache protocols described in this paper use *isochrons* in place of broadcasts to ensure consistent cache updates. The *isochron* is a logically synchronous, sequentially consistent multicast [RWW89]. As in the more concurrent snoopy protocols, writers and readers to the same block can coexist in the delta-cache protocols — a process need not obtain exclusive access rights to a cache block before writing to it. Unlike snoopy protocols, the delta-cache protocols do not require a bus or broadcasting of updates and do not suffer from the severe limitation on scalability caused by bus saturation.

Delta-cache protocols also allow more pipelining than existing directory protocols. Pipelining of memory accesses is important in multiprocessors as a way to reduce the impact of memory latency. When operations may require more than one roundtrip through the ICN to complete, as they do in systems with caches kept consistent with directory protocols, the ability to begin execution of one operation before completion of the preceeding operation is especially important. Finally, the delta-cache protocols are compatible with techniques for executing atomic actions accessing multiple variables in different locations without operations on locks [WiR89]. To the extent other protocols support atomicity, they require writers to obtain exclusive access rights to the accessed block.

Section 2 of this paper defines the cache coherence problem and proposes a new correctness criterion for cache protocols that integrates cache coherence with other aspects of concurrency control and eliminates unnecessary restrictions on pipelining implied by earlier definitions. This correctness criterion is the basis for a simple proof of the delta-cache protocols.

Section 3 defines the network on which delta-cache protocols are based. This network, called an *isotach* network, implements a logical time system in which all operations travel at the same velocity in logical time, one switch per logical time pulse. Isotach networks are feasible. The principal difference between isotach networks and more typical ICN's is that each switch in an isotach network applies a standard list-merge algorithm to the operations arriving on its inputs. We originally proposed isotach networks to support isochrons [RWW89] and later used isotach networks as the basis for concurrency control for systems without caches [WiR89]. This paper extends our work to systems with caches.

Section 4 describes a general delta-cache protocol and two specific protocols based on different patterns for propagating updates. In a delta-cache protocol, caches are out of phase with memory and with each other by a known number of pulses of logical time. The caches are out of phase because propagating cache updates takes time. They are out of phase by a known amount because communication takes a known amount of logical time in an isotach network. The propagation pattern for updates, together with the network topology and size, determines the number of logical time pulses, δ, by which a copy is out of phase with the memory copy. If a PE knows the communication distance to the copy of a variable accessed by a memory or cache operation, it can determine the logical time at which an operation is executed by controlling the time at which it emits the operation. If the PE also knows the δ for the copy accessed by each operation, it can time the emission of a group of operations so the operations are executed at the same logical time, adjusted by the δ for each copy. The PE's can ensure sequential consistency similarly, by controlling the time at which operations are emitted.

Section 5 concludes the paper with a discussion of ongoing and future work.

2. THE CACHE COHERENCE PROBLEM

The cache coherence problem is the problem of maintaining a coherent memory in a system in which processes can write private copies of shared variables. The traditional definition of memory coherence is that a memory is coherent if a load operation on a shared variable returns the value written by the latest store operation on the same variable [CeF78]. Dubois, Scheurich, and Briggs have noted problems with this definition of coherence, in particular, difficulty applying the concept of *latest* store to systems that do not broadcast cache updates [DSB86]. They propose adopting *sequential consistency* as an alternative correctness criterion for cache protocols. An execution is sequentially consistent if it is equivalent to a serial execution in which the operations issued by each process are executed in the order specified by the program [Lam79]. Sequential consistency, and thus memory coherence, can be ensured in systems with caches by prohibiting pipelining of accesses to shared variables [ScD87].

Since pipelining is an important technique for reducing effective memory latency in multiprocessors, researchers have proposed enforcing a weaker form of sequential consistency allowing some pipelining by relaxing ordering constraints between synchronization points [AdH90, DSB86]. With the exception of these protocols, we know of no protocols that allow operations on shared variables to be pipelined. The delta-cache protocols allow most operations to be pipelined and enforce the stronger, original form of sequential consistency.

We view the cache coherence problem as just one aspect of the larger problem of controlling concurrency in parallel computations. In a previous paper [WiR89], we considered the problem of concurrency control in systems without caches. We adapted the definition of *serializability*, the standard correctness criterion in database concurrency control [Pap86], to parallel programs and proposed serializability as the correctness criterion for concurrency control mechanisms for parallel programs. An execution of a parallel program is serializable if operations on shared variables appear to be executed in an order consistent with the program. Sequential consistency is one aspect of serializability, but serializability also encompasses other properties, *atomicity* and *version consistency*. More formally, an execution is serializable if there is an *equivalent* (defined below) serial execution, E_s, with the following properties:

ATOMICITY. The accesses in each atomic action are executed in E_s without interleaving with other accesses.

SEQUENTIAL CONSISTENCY. Accesses by the same process are executed in E_s in the order specified by that process's program, where an order is specified.

VERSION CONSISTENCY. Accesses to the same variable by different processes are executed in E_s in the order specified by the program, where an order is specified.

Two executions of the same program are *equivalent* if each shared variable is accessed by the same operations and each operation returns (in the case of a read operation) or stores (in the case of a write) the same value in both executions.

We can now define memory coherence for multiprocessors with caches:

Memory is *coherent* if for every execution in which one or more operations on a shared variable accesses cache, there is an equivalent serializable execution in which every operation on a shared variable accesses the copy in main memory.

In other words, the presence of caches must not change the meaning of the program. The delta-cache protocols use the isotach network defined in the next section to maintain memory coherence.

3. THE ISOTACH NETWORK

Consider a network of interconnected nodes in which each node is a switch or an element. Each element has a single ICN input and ICN output connected to a switch through a switch interface unit (SIU). In this paper we assume the ICN output for an element is connected to the same switch as the ICN input. An example of such a network is shown in Fig. 1. In the figure, each circle represents a switch and each rectangle an element.

Each SIU emits and receives *messages* on behalf of the associated element. In using the term *message* we do not intend to restrict isotach networks to the message-based programming model. A message is any communication sent from an element to an element. A message may be an operation on a shared variable. A message is *emitted* when the SIU for the source element sends it over the ICN output and is *received* when the SIU for the destination element removes it from the ICN input.

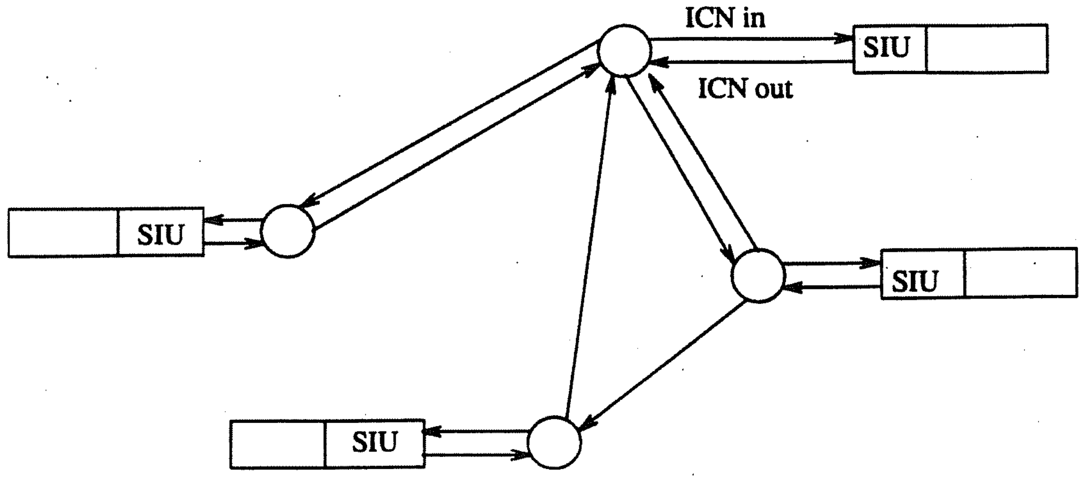

Figure 1. An Example of a Network

With each SIU we associate a local logical clock, a function that assigns monotonically increasing times to local emit and receive events. In general, each logical time is an ordered pair (*pulse,time_within_pulse*), where the first component is an integer and the second an n-tuple of integers. In the logical time system we use in this paper, each logical time is a 3-tuple (*pulse,tick,tock*) of integers. Logical times are lexicographically ordered.

An isotach network implements a logical time system that differs from others [Lam78, Mat88] in that it relates communication time with communication distance. In an isotach network, each message is received *DIST* pulses after it is emitted, where *DIST* is the number of switches through which the message is routed. A message emitted at time $t_{emit} = (i,j,k)$ is received at time $t_{receive} = (i+DIST,j,k)$. An isotach network maintains the following *velocity invariant*:

$$DIST \,/\, (t_{receive} - t_{emit}) = 1 \; switch/pulse$$

All messages in an isotach network travel at the same velocity in logical time — one switch per pulse.

3.1. Implementation

We describe an abstract distributed implementation of an isotach network. The implementation requires that nodes communicate with neighboring nodes over FIFO links. It is based on a form of synchronization we call *local synchrony* (the name is due to Ray R. Wagner, Jr.). Local synchrony has been used by Awerbuch to support execution of SIMD graph algorithms on asynchronous networks [Awe85], by Gibbons to support barrier synchronization [Gib89], and by Ranade in emulating a CRCW-PRAM [Ran87, RBJ88]. Ranade's emulation maintains the velocity invariant and is thus an isotach network. It uses the velocity invariant to support efficient combining of operations within the ICN.

Our implementation of an isotach network is based on the exchange by neighboring switches of control signals called *tokens*. Initially each switch emits a token pulse, i.e., it emits a token on each output. After the initial pulse, each switch emits token pulse i after receiving token $i-1$ on all inputs. Thus each switch is loosely synchronized with its neighbors. The token pulses also drive the clocks at the SIU's. In each token pulse, a switch emits a token on each output, including the output to each adjacent element, if any, and it receives a token on each input, including the input from each adjacent element, before emitting the next token pulse. When it receives token i on its ICN input, the SIU sets its local clock to $(i,0,0)$ and emits the token on the ICN output. The pulse component of the time at each SIU is the number of tokens that have passed through the SIU.

An SIU may emit any number of messages in each pulse. For each message it emits, the SIU determines t_{emit} for the message, timestamps it with t_{emit}, and buffers it for output. The procedure by which the SIU determines t_{emit} depends on the purpose the network is designed to serve. We describe below how the tick and tock components of t_{emit} are determined for a network designed to facilitate memory coherence. The pulse component of t_{emit} is determined by rules called *emission rules* that are at the heart of the delta-cache protocols. The emission rules are discussed section 4.

The SIU emits each message at the time t_{emit} it has stamped on the message. If the pulse component of a message to be emitted is i, the SIU emits the message after it receives and emits the *ith* token and before it receives and emits the *i*+1*st* token. Within each pulse the SIU emits and receives messages in timestamp order. When an SIU emits or receives a message it updates the local clock, setting the clock equal to the timestamp of the message. Since it emits and receives messages in timestamp order, the local clock at each SIU moves forward monotonically. The SIU merges the incoming and outgoing message streams by timestamp, routing the incoming messages to the element and the outgoing messages to the network.

Merging is possible because messages arrive over the ICN input in monotonically increasing order by timestamp. When the SIU receives a message with timestamp (i,j,k), it knows it will receive no message with timestamp less than (i,j,k). Messages are received in timestamp order due to the switch algorithm. Each switch routes messages as usual except it chooses messages to route in timestamp order. A switch with k inputs and j outputs continuously performs a merge of k sorted lists arriving on its inputs producing j sorted output lists. In performing the merge, the switch treats the *ith* token received on an input as having timestamp $(i,0,0)$. When a switch receives a token on each input, the *ith* token received on each input, it consumes the tokens and emits the *i*+1*st* token pulse. After emitting each token pulse it routes the messages received on each of its inputs up to the next token. When it routes a message, the switch increments the pulse component of the message's timestamp. Each time the message is routed, its timestamp is greater than the implicit timestamp for the token emitted before it and less than the implicit timestamp of the token emitted after it. Because each SIU emits messages in timestamp order and the order is maintained at each switch and across each link, each SIU receives messages in timestamp order.

The switch algorithm also ensures the velocity invariant holds. A message with timestamp (i,j,k) arriving at a switch in pulse i (after the *ith* token received on the input on which the message arrives) leaves with timestamp $(i+1,j,k)$ in pulse $i+1$ (after the *i*+1*st* token pulse). Since each switch adds one pulse, a message emitted in pulse i is received in pulse $i+DIST$.

The constraint on the order in which switches route messages can cause communication deadlock in some networks. For this reason, we intend the delta-cache protocols primarily for *equidistant* networks in which PE's are connected to MM's via a multistage interconnection network (MIN). Deadlock is easily avoided in equidistant networks, but in a nonequidistant network, e.g., a hypercube, insufficient buffer space at the switches can lead to deadlock. In such networks, a switch may need to buffer a message to release a channel along which a second message must be routed before the first can proceed. In this paper we assume infinite buffers at the switches in the ICN of nonequidistant networks. Finding realistic ways to avoid communication deadlock in nonequidistant networks is a current topic of research.

3.2. A Shared Memory Model Isotach Network

We now adapt this abstract implementation of an isotach network to asynchronous shared memory model computations. Each element node in the network contains an SIU and either a memory module (MM) or a processing element (PE). Each element with a PE may also contain a cache. Messages in the shared memory model are primarily operations and responses to operations, including cache updates. An *operation* is an instruction accessing a shared variable. Initially, we assume operations are of two types: reads and writes. Each operation is issued by processes under control of the process's program. For simplicity we assume each process executes on its own PE. Messages may also be commands issued by the PE or cache under control of the operating system or cache protocol.

We distinguish the issuing of a message from the emission of the message. A message is *issued* when is generated by the MM, PE, or cache and put under the control of the local SIU. We require that processes issue operations in the order specified by the program. As a consequence, each SIU acquires control of locally issued operations in the order specified by the program, possibly interspersed with other locally generated messages.

The procedure for determining t_{emit} for a message is designed to facilitate concurrency control. We distinguish two types of messages for the purpose of determining t_{emit}: primary and secondary messages. The issuance of a secondary message is prompted by the emission or receipt of another message and the time this other message is received or emitted fixes t_{emit} for the secondary message. The most common type of secondary message is the *response*. A response is a message issued as part of the processing of another message and must be issued and emitted at the same logical time as the processing of the message that prompts its issuance. Responses include cache updates and the values returned by memory in response to read operations. Though the message prompting the issuance of a secondary message may itself be a secondary message, every secondary message can be traced back to a primary message. A primary message can be emitted in any pulse subject to the constraints of atomicity and sequential consistency as expressed by the emission rules. Primary messages are typically operations. Each operation is initially a primary message, but in some cache protocols an operation becomes a secondary message before it is executed. If an MM processes an operation by forwarding it to a cache for execution instead of executing the operation itself, the message the MM sends to the cache is a response and is emitted by the MM at the same logical time as it processes the operation. We assume MM's and caches issue only secondary messages. PE's may issue both primary and secondary messages.

For primary messages, the tick component is the *pId* of the issuing PE. The tock component is the local issue order of the message, the position of the message in the sequence of locally generated primary messages. A message has tock component *k* if it is the *kth* primary message issued locally during program execution. When t_{emit} is assigned in this way, the velocity invariant ensures operations received in the same pulse are received in increasing order by the *pId* of the issuing PE and operations received in the same pulse and emitted by the same PE are received in the order in which they were issued.

For secondary messages, the tick and tock component of t_{emit} are the tick and tock component of the emit or receive event prompting the issuance of the message. When it receives a message, the SIU sends the message to the destination MM, PE, or cache and waits for a response or other acknowledgement before receiving or emitting the next message. Hence each message is processed at the same logical time as it is received and each response is issued and emitted at the same logical time as the receipt and processing of the message that prompts its issuance. Since cache updates are responses, cache updates received in the same pulse are received in increasing order by the *pId* of the PE that generated the operation the update reflects. Updates received in the same pulse reflecting operations issued from the same PE are received in the order in which they are issued.

The abstract implementation described here is intended to be useful in proving serializability, but is not the implementation we recommend for an actual system. An implementation need only ensure each execution is equivalent to some execution based on the abstract implementation, i.e., that operations on each variable are executed in the same order in both executions. Token pulses are necessary, but not clocks. Timestamps are necessary, but the pulse component can be omitted and the tock (issue order) component abbreviated. The tock component need not be the issue order since the beginning of execution, potentially a large integer, but need only specify the issue order of the operation among those operations issued by the same PE that access the same variable with the same effective execution pulse. (The concept of effective execution pulse is defined in section 4 below.) By bounding the number of such operations to a small constant, the size of the tock component can be bounded. Since the tick component, the *pId* for the issuing PE, is typically needed for reasons unrelated to the velocity invariant, e.g. as a return address for responses, the additional message size required to implement an isotach network is small. The additional time required to maintain the velocity invariant in an isotach network is more difficult to determine and is the subject of current research.

3.3. Isochrons

An isochron is a logically synchronous, sequentially consistent multicast. In the shared memory programming model, an isochron consists of operations on shared variables. Operations within each isochron are executed atomically in the order in which they are listed in the isochron and isochrons issued by the same process are executed in the order specified by the program. When isochrons are implemented using an isotach network, operations can be executed out of order relative to a global physical clock and execution of operations from different isochrons can be interleaved. The implementation nevertheless preserves the semantics of isochrons. Each execution is equivalent to an execution in which isochrons are executed atomically and in order.

Syntactically, an isochron is a list of one or more operations terminated by a semicolon in which adjacent operations are separated by double bars "||." For example, consider the following code segment:

```
        A:read(a) || B:read(b) || B:write(5);
        A:write(b);
```

The first isochron assigns the value of shared variables A and B to local variables *a* and *b*, respectively, and assigns the value 5 to B. The second isochron assigns to A the value returned by the read on B in the first isochron. The operations within each isochron are executed atomically. For example, no process can change the value of B between the read and write operation on B. Such interleaving of accesses by different processes may not occur within an isochron but may occur between isochrons. Another process may change the value of B before the write on A. Execution is sequentially consistent in that the read operation on B is executed before the write and the first isochron will appear to be executed before the second. In particular, no process that reads the *new* value assigned to A in the second isochron will read the *old* value of B, before the write in the first isochron, in a subsequently issued read operation.

The implementation of isochrons is based on the velocity invariant maintained by an isotach network. If a PE-SIU knows *DIST*, the distance an operation travels to memory, the PE-SIU can control $t_{receive}$ by its choice of t_{emit}. This control is the basis for implementing isochrons. A PE-SIU emits operations issued by the associated process in accordance with the following emission rules:

ATOMICITY. Emit operations from the same isochron so they are received (and thus executed) in the same pulse.

SEQUENTIAL CONSISTENCY. Emit each operation so it is received in a pulse equal to or greater than that of the operation issued before it.

The sequential consistency rule is sufficient because operations issued by the same process and received in the same pulse are received in the order in which they are issued (in increasing order by tock component). For equidistant networks, application of the emission rules is especially simple. A PE-SIU emits all operations from the same isochron in the same logical pulse and emits operations in the order in which they are issued by the associated process.

These rules ensure serializable execution for a limited class of programs called isochron programs. An isochron program is a program that is serializable if it is sequentially consistent and every isochron is executed atomically. The emission rules ensure for every execution E_p of an isochron program, there is an equivalent serial execution E_s in which all the operations in each isochron are executed without interleaving and operations issued by the same process are executed in order. Execution E_s is the execution in which operations are executed serially in the order of the logical time at which they are received in E_p.

In the next section we adapt the emission rules to multiprocessors with caches and extend the class of programs for which the rules ensure serializable execution.

4. DELTA-CACHE PROTOCOLS

We consider a Δ-stage MIN-based multiprocessor in which each PE has a local cache that may contain shared variables. The MIN is an isotach network maintaining the velocity invariant for all messages. The delta-cache protocols rely on the fact that the velocity invariant applies to cache updates to ensure cache copies remain out of phase with memory by a fixed number of logical pulses.

The local PE-SIU ensures the cache receives operations in the correct order. Every message received by the cache goes through the local PE-SIU. The structure of a node containing a PE, PE-SIU, and cache is shown in Fig 2. The cache can send messages directly to the PE but all messages from the PE to the local cache go through the PE-SIU.

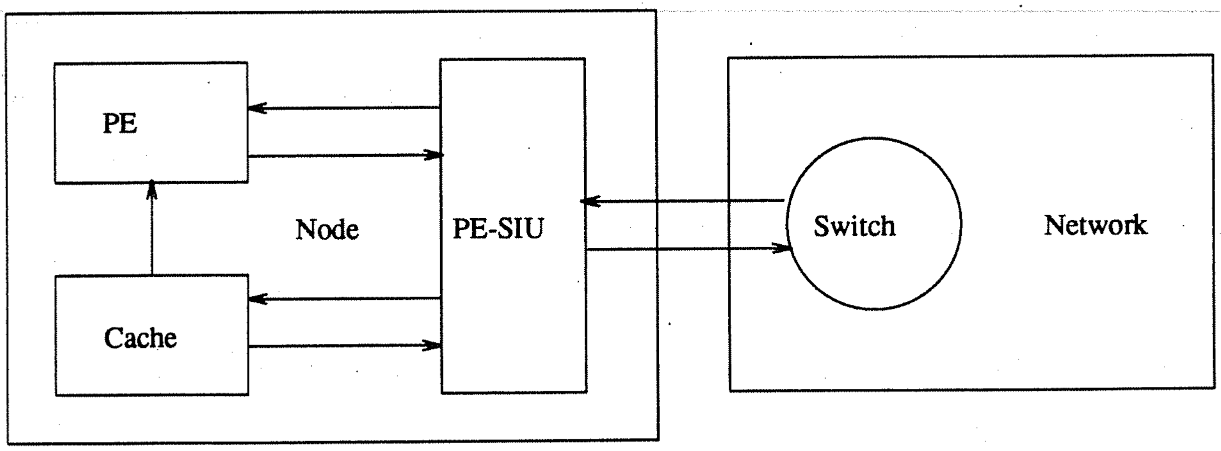

Figure 2. PE Node Block Diagram

A message received by the cache is *local* if it is sent to the cache by the local PE. A local message goes through the PE-SIU but not through the network. All other messages received by the cache are *incoming* messages. Incoming messages are received by the cache from the network through the SIU. A cache update received from the network is an incoming message even if the write that prompted the update was issued by the local PE.

As described above in the section on implementing isotach networks, each PE-SIU merges locally generated messages with the stream of incoming messages by timestamp. As a consequence, the cache receives messages in order by $t_{receive}$. Because a local message is not routed through any network switch, the velocity invariant implies $t_{emit} = t_{receive}$. In other words, a local operation is received by the PE-SIU at the same logical time as it is emitted. Note that $t_{receive}$ for a local message emitted by the SIU for PE_j in pulse i is (i,j,k), where k is the issue order of the message. The merging of the incoming and local messages by $t_{receive}$ ensures the velocity invariant applies to local messages as well as to messages that traverse the ICN.

Except in a special case described below, the delta-cache protocols use a write-through (also called a write-update) policy, i.e., each write to a cache copy updates all other copies, including the main memory copy. Under the alternative write-back (write-invalidate) policy, a write to a cache copy invalidates all other copies. Main memory is updated only when the cache block is recalled or evicted. Because the write-back policy cannot support multiple readers and writers, delta-cache protocols use the write-through policy. A disadvantage of the write-through policy is the cost of distributing cache updates. Several schemes to reduce this cost on equidistant networks [Ste89] are compatible with isotach networks.

For simplicity, we assume each process executes on its own processor and each cache block contains exactly one variable. Thus we use the terms *cache block* and *variable* interchangeably. The home MM, the MM containing the memory copy of the cache block, maintains a directory *DIR* for each block recording the *pId*'s of all PE's in the *cacheSet*, the set of PE's with a cache copy of the block. For simplicity, we use the bit vector representation for directories proposed by Censier and Feautrier [CeF78] in which bit k of the vector is set if $PE_k \in cacheSet$.

The delta-cache protocols use the isotach network to maintain *δ–invariants.* A δ-invariant relates the value of a cache copy of a variable to the value of the variable in main memory. For a given cache copy of a given variable V, a δ-invariant is of the form

$$V_{cache,i} = V_{memory,i+\delta}$$

i.e., the value of the cache copy of variable V at logical time (i,j,k) equals the value of V in main memory at logical time $(i+\delta,j,k)$. The value of δ is different for different network topologies and for different classes of delta-cache protocols.

Delta-cache protocols can be classified as *late*, *early*, or *on–time* protocols, respectively, depending on whether the value of a variable in cache lags behind, is up to date with, or is ahead of the main memory copy. In a late cache protocol each write is executed at the home MM and propagated by the home MM to the caches. For a Δ-stage MIN, δ is –Δ. In other words, the protocol maintains the invariant

$$V_{cache,i} = V_{memory,i-\Delta}$$

In an on-time protocol, the caches and home MM execute each write in the same logical pulse. Since cache copies are up to date with the memory copy, δ is zero. In an early cache protocol one cache copy, the *hot* copy, is Δ pulses ahead of the memory copy. For the cache with the hot copy, δ is Δ. In this paper, we describe late and early cache protocols but leave discussion of on-time protocols for later work.

In nonequidistant networks, the logical time required to propagate updates is nonuniform and δ may differ for different variables in the same cache and different copies of the same variable. For example, in a late cache protocol the caches furthest from the node containing the home MM have the most out of date cache copies and thus the largest absolute δ values. Even on equidistant networks, δ may be different for different variables. Different delta-cache protocols may be used for different variables in the same execution for performance reasons. If a late cache protocol is used for variable A and and an early cache protocol for B, different δ values will apply to A and B.

The emission rules for systems with caches are the same as for systems without caches except the rules apply to the pulse in which operations are *effectively* executed rather than the pulse in which operations are *actually* executed. The *effective* pulse *EP* for an operation is the execution pulse of the *equivalent* operation on the memory copy. If the *EP* for an operation executed at (i,j,k) is i', the execution of the operation is equivalent to execution of the same operation on the memory copy at time (i',j,k). If an operation is executed at memory, its effective and actual execution pulses are the same. The *EP* for a read operation executed at memory is the pulse in which the operation is executed. The *EP* for a write operation is the pulse in which the value written is assigned to the memory copy. If the write is executed at memory, the *EP* is again the pulse in which the operation is actually executed. We distinguish the execution of a write operation from the execution of the updates prompted by the write operation. As a reminder of this distinction, we say operations are *executed* and updates *performed*. If the write is executed on a cache copy and the value written is assigned to memory as the result of an update prompted by the write, the *EP* is the pulse in which the update is performed. For a read executed on a cache copy, the *EP* depends on the δ-invariant. By the δ-invariant, a read executed on the cache copy at time (i,j,k) returns the same value as a read executed on the memory copy at time $(i+\delta,j,k)$. Hence the *EP* for a read executed on a cache copy in pulse i is $i+\delta$.

Given an execution E_c in which one or more operations are executed on cache copies, consider the execution E_p constructed from E_c by replacing each operation with the equivalent operation executed at memory. For any given operation OP_x executed at (i,j,k) in E_c, OP_x is executed in E_p at (i',j,k), where i' is the *EP* for OP_x. Assuming the δ-invariant holds, each read or write in E_c returns or stores the same value as the corresponding operation in E_p.. Hence E_c and E_p are equivalent. By definition, memory is coherent if E_p is serializable.

To ensure E_p is serializable, each PE-SIU emits operations in accordance with the emission rules as adapted for systems with caches:

ATOMICITY. Emit operations from the same isochron so that the *EP* for each operation is the same.

SEQUENTIAL CONSISTENCY. Emit each operation so its *EP* is no less than that of the operation issued before it.

Note the emission rules for systems without caches are a special case of these rules. Assuming the δ-invariant holds, the emission rules imply memory coherence for isochron programs. For each execution E_c of an isochron program P in which one or more operations access cache, there is an equivalent serializable execution E_p of P in which all operations access main memory.

In the remainder of this section we first describe individual delta-cache protocols and then show how to extend the class of programs for which the protocols ensure memory coherence.

4.1. A Late Cache Protocol

We describe a late cache protocol for a Δ-stage MIN-based multiprocessor. A late cache protocol for this topology maintains the δ-invariant

$$V_{cache,i} = V_{memory,i-\Delta}$$

The cache copy of any shared variable `v` at time (i,j,k) equals the memory copy of `v` at $(i-\Delta,j,k)$.

The protocol recognizes four types of operations on any shared variable `v`. An `r′` is a read and a `w′` a write emitted by a PE with a cache copy of `v`. An `r` is a read and a `w` a write emitted by a PE with no cache copy of `v`. Each write operation, whether it is a `w` or `w′`, is executed at the home MM. An MM executes a `w` or `w′` on `v` by updating the memory copy, adding the source PE to the directory *DIR* for `v`, if it is not already in *DIR*, and emitting a cache update to all PE's ∈ *DIR*. An `r` is executed by the home MM and an `r′` is executed locally, on the cache copy. We refer to `w`, `w′`, and `r` operations as memory operations and `r′` operations as cache operations. We also define a `release` operation. A `release` on `v` emitted by PE_i instructs the home MM to remove PE_i from *DIR*. One reason a PE may emit a `release` is to free cache space for another block.

Schemata for executing operations in the late cache protocol are given in Fig. 3. In the schema for each operation, the circle represents the execution of the operation and the rectangle, the *EP* for the operation. Each triangle represents the updates sent by the MM to all PE's $\in$ *DIR*. The label "M" indicates execution at the home MM. Each horizontal line represents a logical time and the distance in logical time between time lines is Δ pulses. For example, the schema for the w' operation shows a w' is executed at the home MM Δ pulses after it is emitted, the *EP* is the same as the pulse in which it is executed, and the updates prompted by the w' are received Δ pulses after the w' is executed.

The δ-invariant holds because each write operation is executed on the memory copy Δ pulses before the cache copies are updated. Note the cache copy of the processor emitting a write is updated at the time the processor receives the update from the MM, not at the time the write is emitted. When it executes a write, an MM sends updates to all PE's $\in$ *DIR*, including the PE that emitted the write. (To reduce the number of updates, the PE emitting a `w` could buffer the value written and update its cache without prompting from memory 2 Δ pulses later.)

A PE obtains a cache copy of variable `v` Δ pulses after memory executes the PE's initial operation on `v`. The value of the cache copy at time t, when the PE obtains the cache copy, is the value of the memory copy Δ pulses before t. The δ-invariant continues to hold until execution ends or until Δ pulses after memory executes the PE's `release` on `v`. The MM sends the PE an update for each write the MM executes on `v` between the PE's initial operation and the `release` and the update is received and per-

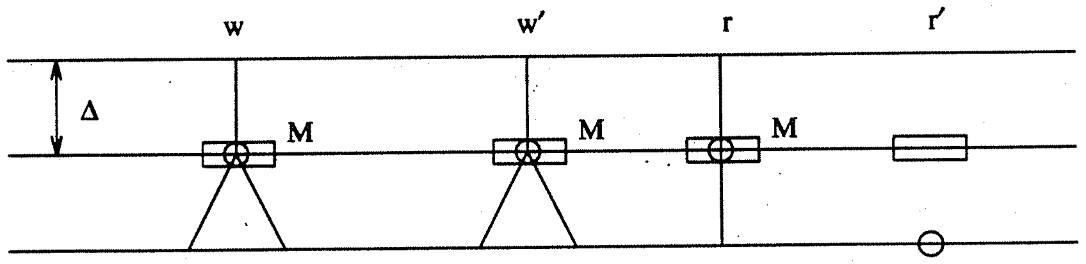

Figure 3. Schemata for Late Cache Protocol.

formed on the cache copy Δ pulses after the corresponding write is executed on the memory copy. Since the cache copy of `V` remains valid, a PE may continue to emit `r'` operations on `V` until 2 Δ pulses after it emits a `release` on `V`. If a PE emits a `release` on `V` and subsequently emits a memory operation accessing `V`, the PE again obtains a cache copy 2 Δ pulses after it emits the memory operation.

The *EP* for a memory operation is Δ pulses after it is emitted and the *EP* of a cache operation is Δ pulses before it is emitted. Let $\rho_{x,i}$ denote the pulse in which operation OP_i is executed and $\rho_{s,i}$ the pulse in which OP_i is emitted. (The second subscript is omitted unless needed for clarity.) Recall that for a write operation, the *EP* is the pulse in which the memory copy is assigned the value written. In the late cache protocol all writes are executed at memory, so $EP = \rho_x = \rho_s + \Delta$. Since an `r` is executed at memory, the *EP* for an `r` is also $\rho_x = \rho_s + \Delta$. For an `r'` operation, $EP = \rho_x - \Delta$. By the δ-invariant, an `r'` is equivalent to a read executed at memory Δ pulses before the `r'` is executed at cache. Since $\rho_x = \rho_s$ for a cache operation, the *EP* for an `r'` can also be expressed as $\rho_s - \Delta$.

If a PE-SIU emits operation OP_c 2Δ pulses after OP_m, where OP_c is a cache operation and OP_m is a memory operation, the *EP* for both operations is $\rho_{x,m}$. (For OP_c, $EP = \rho_{s,c} - \Delta = \rho_{s,m} + 2\Delta - \Delta = \rho_{x,m}$.) In general, a PE-SIU ensures atomic execution of isochrons by emitting operations in the same pulse if all or none of the operations are cache operations and emitting cache operations 2Δ pulses after memory operations otherwise. In either case, the *EP* is the same for all operations in the isochron.

As an example of the application of this rule, consider the following isochron program in which processes P_i, P_j, and P_k read or write shared variables `A` and `B`.

P_i::	`A:write(0) || B:write(0);`	*assign 0 to A and B atomically*
P_j::	`A:read(a) || B:read(b);`	*read A and B atomically*
P_k::	`A:write(1) || B:write(1);`	*assign 1 to A and B atomically*

We assume processes P_i and P_j have cache copies of `A`, but not `B`. Process P_k has a cache copy of `B` but not `A`. Fig. 4 shows one possible execution of this program. In the figure, each horizontal line represents a logical time. A time line labeled *l* represents a logical time in which the tick (*pId*) component is *l*. The distance between adjacent pairs of time lines with the same label is Δ. As shown by the rectangles

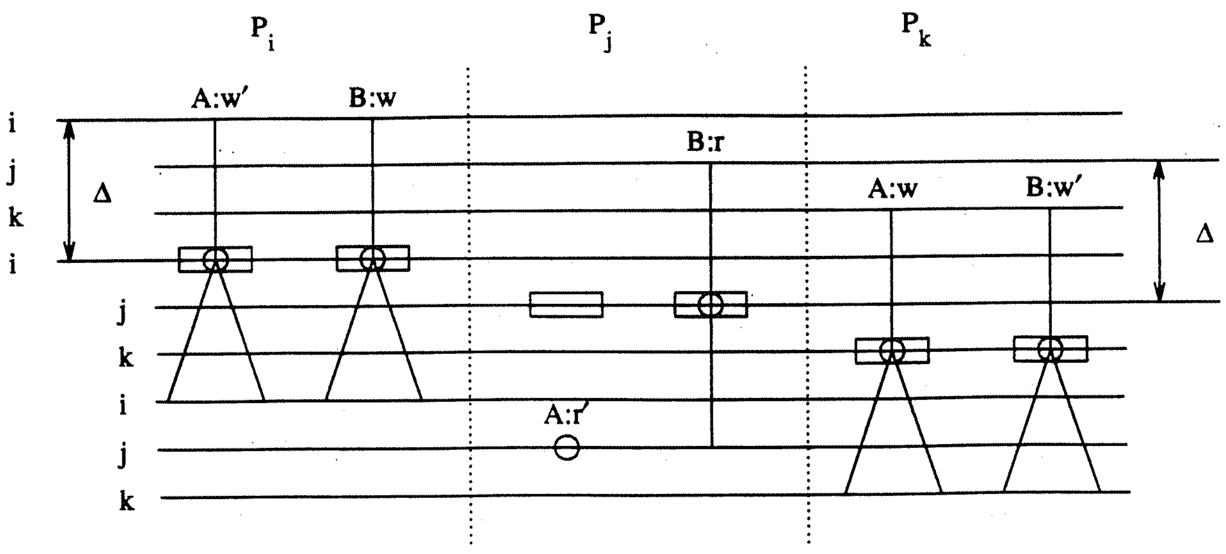

Figure 4. Atomic Execution under Late Cache Protocol

representing *EP*'s, each isochron is executed atomically and the execution is equivalent to the serial execution of P_i, P_j, and P_k's code, in that order. Note the protocol permits multiple concurrent readers and writers to obtain consistent views and make consistent updates to the same variables.

The emission rule for sequential consistency implies a cache operation issued after a memory operation can be emitted no earlier than 2Δ pulses after the memory operation. The delay gives the cache operation an *EP* no less than the preceding memory operation. On the other hand, a memory operation issued after a cache operation can be emitted as many as 2Δ pulses before the cache operation and still appear to be executed after the cache operation. As an example of this rule, consider the following program in which processes P_i and P_j access shared variables `A` and `B`:

```
Pi::  A:write(2); B:write(2);          assign 2 to A then assign 2 to B
Pj::  B:read(b); A:read(a);            read B then read A
```

We assume both P_i and P_j have cache copies of `A` and neither has a cache copy of `B`. An unserializable execution of this program resulting from pipelining a cache operation, P_j's `r′` on `A`, is shown in Fig. 5. The execution is unserializable because an execution in which P_j reads the *old* value of `A`, the value before P_i's write on `A` is executed, after reading the *new* value of `B` is not equivalent to any serial execution in which the operations issued by each process are executed in the order specified by the program. The incorrectly emitted operation is marked in the figure with a question mark. The r' by P_j on A should have been emitted no earlier than the time indicated by the exclamation mark.

The rule restricting the pipelining of cache operations following memory operations can be relaxed for an r' closely following a write to the same variable emitted by the same PE. If the PE-SIU buffers the value stored by a write operation while awaiting the corresponding update from MM, it can immediately satisfy any subsequent r' on the same variable with an EP equal to the EP of the write by returning the buffered value. Buffering the value to be written allows the cache to pipeline execution but does not change the EP for either the read or the write.

Late cache protocols can also be devised for nonequidistant networks, assuming the communication distance between every pair of nodes is known. For simplicity, we assume the distance between any given pair of nodes is constant and the same in both directions. Let d_i be the distance between the node emitting operation OP_i and the node containing the memory copy of the variable accessed by OP_i. Assuming updates are issued as before, δ for the cache copy of the variable at the node emitting OP_i is $-d_i$. The PE-SIU's can ensure memory coherence for isochron programs by following the emission rules for caches. For example, for cache operation OP_c, $EP = \rho_{x,c} - d_c$ and for memory operation OP_m, $EP = \rho_{x,m} = \rho_{s,m} + d_m$. A PE-SIU ensures atomic execution of OP_c and OP_m by emitting the cache operation

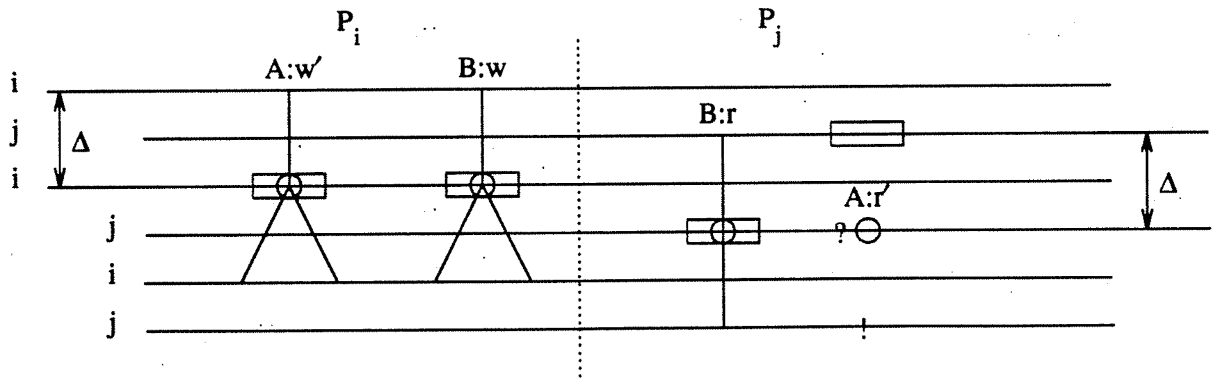

Figure 5. An Incorrect Execution

$d_c + d_m$ pulses after the memory operation. The *EP* of both operations is then $\rho_{x,m}$.

4.2. An Early Cache Protocol

In an early cache protocol, one copy, the *hot* copy is δ pulses ahead of the other copies. The PE with the hot copy of variable `v`, called the *owner* of `v`, has special, but not exclusive, access rights to `v`. Processors other than `v`'s owner may both read and write `v`. To accommodate changing reference patterns, the protocol allows the hot copy to migrate. Different variables may be owned by different PE's and the same variable may be owned by different PE's at different times during execution.

The early cache protocol we describe here is for a MIN-based multiprocessor in which the communication distance between every PE-PE pair is Δ and is the same as the distance between every PE-MM pair. A simple way to construct a multiprocessor with this topology is to connect each output from the last stage of a $\Delta-1$ stage ICN to the corresponding input to the first stage and attach either a PE or an MM to each switch in the first stage. Versions of the protocol described here can be devised for other multiprocessor topologies including those described in the previous section.

A cache copy other than a hot copy is called a *cold* copy. The protocol maintains, for any shared variable `v`, the following δ-invariant:

$$V_{hot,i} = V_{memory,i+\Delta} = V_{cold,i+\Delta}$$

The hot copy of `v` at time (i,j,k) equals the memory copy and cold copies of `v` at $(i+\Delta,j,k)$.

4.2.1. Static Version

Initially, we assume the hot copy is stationary. Later we describe a dynamic version in which the hot copy may migrate. The owner maintains the directory *DIR* for `v`. The home MM records `v`'s owner, but does not have a copy of *DIR*. The protocol recognizes two types of operations on any variable `v` in addition to those defined for the late cache protocol: an `r''` is a read and a `w''` a write emitted by `v`'s owner. In this protocol, `w'` and `r'` denote operations emitted by a PE with a cold copy. A PE with a cold copy may emit a `release` operation. A `release` on `v` is sent directly to `v`'s owner. The owner executes a `release` by deleting the source PE from *DIR*.

Each `r` and `w` operation on `v` is sent to the home MM. The home MM forwards the `r` or `w` to the owner but does not execute the operation. Each forwarded operation is a response, i.e., the MM forwards the operation at the same logical time as it receives the operation. As a result, each `r` and `w` on `v` is received by the owner of `v` exactly 2Δ pulses after it is emitted. The requirement that `r` and `w` operations be routed through the home MM is imposed in preparation for the dynamic version of this protocol. In the dynamic version, a PE with no cache copy of `v` must route operations on `v` through the home MM because the PE does not know the location of the hot copy. A PE with a cold copy of `v` knows `v`'s owner and can send `w'` and `r'` operations directly to the owner.

Each write operation, whether it is a `w`, `w'`, or `w"`, is executed by the owner. When it executes a write on `v`, the owner assigns the value written to the hot copy, adds the source PE to *DIR* if it is not already in *DIR*, and sends updates to the home MM and to all PE's $\in$ *DIR*. With each update it sends its own *pId*. The owner also executes each `r` and `r"` on `v`. It executes an `r` operation by adding the source PE to *DIR* and returning a copy of `v` and its own *pId*. The only operation on `v` not executed by the owner is the `r'`. An `r'` is executed on the local cold copy by the PE that emits the operation. The schemata in Fig. 6 summarize the protocol. The dashed lines represent steps required during migration of the hot copy and are discussed later in this section. Execution at the hot copy is indicated by the label "H."

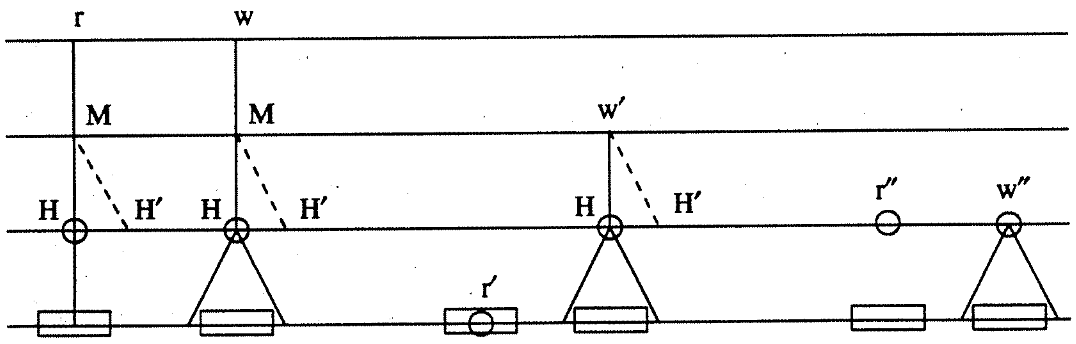

Figure 6. Schemata for Early Cache Protocol

Since every read is executed on a cache copy, the memory copy can be eliminated. Eliminating the memory copy allows an optimization for the special case in which only the owner has a cache copy. Studies of parallel programs suggest this case occurs frequently in actual applications [BaR89, EgK88]. The owner can detect whether it has the only cache copy because it maintains *DIR*. If *DIR* is empty, the owner does not use a write-through policy, but instead executes all operations on `v` locally. The memory copy is also not required in the dynamic version of this protocol, but, because it is useful in reasoning about the protocol, we will continue to assume the home MM maintains a copy of `v`.

The memory copy of `v` lags Δ pulses behind the hot copy because every value written to `v` is assigned to the hot copy first and to the memory copy Δ pulses later. Every cold copy of `v` is up to date with the memory copy because the updates reflecting each write to `v` are performed at memory and at all PE's with a cold copy of `v` at the same logical time. A new cold copy of `v` is created whenever a PE with no cold copy accesses `v`. The PE acquires the cold copy 3Δ pulses after emitting the `w` or `r` on `v`. When first acquired, the cold copy is Δ pulses behind the *hot* copy since it is sent by the owner Δ pulses earlier. Since the memory copy is also Δ pulses behind the hot copy, a new cold copy initially equals the memory copy. Thereafter, until 2Δ pulses after the PE with the cold copy emits a `release` on `v`, each update to `v` is performed at the cold cache at the same logical time as at memory. The protocol thus maintains the δ-invariant.

Each PE-SIU determines the *EP* for each operation it emits. The *EP* for each of the six types of operations relative to ρ_x and ρ_s is given in Table 1. Recall ρ_x denotes the pulse in which an operation is executed and ρ_s the pulse in which an operation is emitted. Since the *EP* for a write is the pulse in which the value written is assigned to the memory copy and the value written is assigned to the memory copy in this protocol Δ pulses after the write is executed, the *EP* for all write operations is $\rho_x + \Delta$. The *EP* for reads is determined by the δ-invariant. Since `r` and `r″` operations are executed on the hot copy and the memory copy lags Δ pulses behind of the hot copy, the *EP* for these operations is also $\rho_x + \Delta$. The *EP* for an `r′` is the same as the execution pulse since every cold copy equals the memory copy at the same logical time.

operation	ρ_x	ρ_s
r	$+\Delta$	$+3\Delta$
w	$+\Delta$	$+3\Delta$
r′	$+0$	$+0$
w′	$+\Delta$	$+2\Delta$
r″	$+\Delta$	$+\Delta$
w″	$+\Delta$	$+\Delta$

Table 1. Relating ρ_x, ρ_s, and *EP*

As in the late cache protocol, the PE-SIU's ensure memory coherence for isochron programs by emitting operations in accordance with the emission rules, applied to the *EP* of each operation. For example, if PE_j has a hot copy of `A`, a cold copy of `B`, and no cache copy of `C`, it executes

```
A:read(a) || B:read(b) || C: read(c);
```

by emitting the `r″` on `A`, the `r′` on `B`, and the `r` on `C` so that the *EP*'s are the same. Assuming PE_j emits the `r` in pulse i, it emits the `r″` in pulse i+2Δ and the `r′` in pulse i+3Δ. If another processor PE_i concurrently writes `A`, `B`, and `C` and PE_i also conforms to the emission rules, PE_j will get a consistent view, i.e., the values returned will either all be the *old* values, before execution of the writes by PE_i, or all will be the *new* values written by PE_i. Fig. 7 is a schematic diagram of one possible execution of PE_i and PE_j's isochrons. We assume PE_i has the hot copy of `B`, a cold copy of `C`, and no copy of `A`. The execution shown is equivalent to a serial execution of PE_j's isochron followed by PE_i's.

A PE can pipeline an operation if the resulting *EP* is no less than the operation issued before it. For example, if a PE emits an `r` or `w` at time t, it cannot emit an `r′` for 3Δ pulses after t. On the other hand, an `r` issued after an `r′` can be emitted 3Δ pulses before the `r′` and still appear to be executed after the `r′`. The restriction on pipelining implied by the emission rules can be relaxed in the case of a read following a write to the same variable by using the buffering technique described above in relation to the late

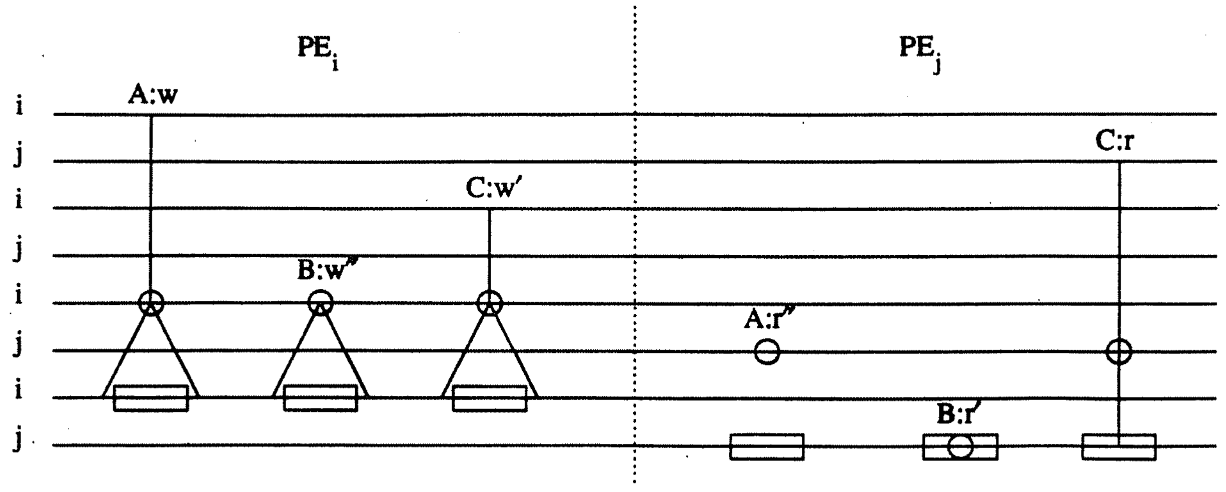

Figure 7. Atomic Execution under Early Cache Protocol

cache protocol.

The early cache protocol supports a broader class of atomic actions than the late cache protocol using the techniques described so far. For example, a late cache protocol cannot, using only isochrons, support atomic execution of the assignment `A := B`. Isochrons by themselves support only a limited class of atomic actions. The PE emitting an isochron must be able to emit all the operations as a batch. Since the PE cannot emit the write on `A` in the assignment `A := B` until after it receives the response to the read on `B`, isochrons are insufficient to support atomic execution of this assignment. In a later section we describe techniques that extend the power of isochrons in executing atomic actions.

The early cache protocol provides an alternative means for executing atomic actions similar to the traditional locking approach. A PE can execute any atomic action if it owns the variables accessed in the atomic action. For example, a PE can execute `A := B`, where it owns both `A` and `B`, by emitting an `r″` on `B`, waiting until cache responds, and then, still in the same pulse, emitting a `w″` on `A` that assigns the value returned from cache.

This technique is similar to locking in that the PE must own all the variables accessed, but there is an important difference. Ownership does not imply exclusive access. A PE can both read and write variables owned by another PE and can even access the variables atomically if the atomic action can be expressed as an isochron.

Using ownership as the basis for executing atomic requires the ability to change ownership dynamically. This ability is also desirable as a way to allow the hot copy of a variable to migrate to the PE most intensively accessing the variable. In the next section we consider changing ownership dynamically.

4.2.2. Dynamic Version

We describe an algorithm, one among several possible algorithms, for changing the location of the hot copy. The algorithm we describe maintains the δ-invariant and the *EP* for each operation. As a consequence, processes can continue to access V while the hot copy of V is in motion.

The hot copy moves in response to explicit requests. A PE requests ownership by sending a *request* operation to V's home MM. The request queue for V is represented as a linked list distributed among the PE's on the queue. For each block it contains, the MM records the *pId* of the PE that emitted the last *request* for the block, thus maintaining a pointer to the tail of the request queue. When it receives a *request* for V, the MM forwards it to the PE currently at the tail of the queue and then updates the tail. Thus each PE in the request queue, except the PE at the tail, knows its successor.

The owner of V can relinquish the hot copy at any time after it receives the *pId* of its successor on the request queue. We assume the hot copy of any variable V is always located at a PE. (Alternatively, the home MM could be the default owner.) At initialization, the hot copy of V is located at a PE (ideally the first PE to access V) and no PE relinquishes ownership until another requests it.

Let H denote the current owner of V, V_H the copy of V at H, H′ the new owner (the successor of H on the request queue), and $V_{H'}$ the copy of V at H′. H initiates the migration of the hot copy to H′ by emitting a change of address "COA" multicast naming H′ as the owner. Let t_r denote the logical time at which H emits the COA. We assume $H' \in DIR$ when the migration begins and no releases on V or operations on V by H or H′ are executed concurrently with the migration. We also assume no PE emits a w′ on V until at least Δ pulses after it receives a new cold copy of V. (We eliminate these assumptions later in this section.) The steps in moving the hot copy of v, summarized in Fig. 8, are as follows:

Time t_r

H initiates the migration algorithm by sending a COA multicast to all PE's $\in$ *DIR* and to the home MM.

Interval t_r to $t_r+\Delta$

H executes operations as before t_r except H identifies H′ as the owner of v in response to any r or w operation.

Time $t_r+\Delta$

The home MM and all PE's $\in$ *DIR* receive the COA.

Interval $t_r+\Delta$ to $t_r+2\Delta$

H executes operations as in the interval t_r to $t_r+\Delta$. In the Δ pulses after they receive the COA, the home MM and PE's $\in$ *DIR* send any r, w, and w′ operations on v to both H and H′. The sending of operations to H′ during this interval is represented in Fig. 6 by the dashed lines.

Time $t_r+2\Delta$

The home MM and PE's $\in$ *DIR* substitute H′ for H as the owner of v and send all further operations on v only to H′. H allocates a new copy of v, denoted v_{cool}, initially equal to v_H, substitutes itself for H′ in *DIR* and sends a copy of *DIR* to H′. H′ allocates a new copy of v, denoted v_{warm}, initially marked undefined, and creates a new interim directory *DIR′*.

Interval $t_r+2\Delta$ to $t_r+3\Delta$

H continues to execute operations on v as in the Δ pulses after t_r. Each r, w, and w′ operation executed at H during this interval is executed at the same logical time at H′. The execution by H′ is local — H′ does not send updates or replies. When it receives a w or w′, H′ assigns the value written to v_{warm} (but not $v_{H'}$), marking v_{warm} as defined. When it receives a w or r, it adds the source PE to *DIR′*. H′ continues to receive cold copy updates from H. It performs the updates only on $v_{H'}$.

Time $t_r+3\Delta$

H makes v_H a cold copy by setting it equal to v_{cool}. H discards v_{cool} and *DIR*. H′ makes $v_{H'}$ the hot copy, setting it equal to v_{warm} if v_{warm} is defined. H′ discards v_{warm}.

All events that the migration algorithm specifies occur at $k*\Delta$ pulses after t_r occur exactly $k*\Delta$ pulses after t_r, i.e., in a different pulse but the same tick and tock as t_{emit} for the COA.

Fig. 9 shows operations on v emitted by PE's other than H and H′ and executed while the hot copy of v is migrating. Executions of r′ operations are unaffected by the migration. Each r, w, and w′ is executed according to its *EP* as follows:

	H	H′	MM and PE's $\in$ DIR
t_r	send COA		
⋮			
$+\Delta$		receive COA	receive COA
⋮			MM sends r &w and PE $\in$ DIR send w′ to both H and H′
$+2\Delta$	DIR := DIR-H′+H send DIR to H′ allocate $V_{cool} := V_H$	create DIR′=$\varnothing$ allocate V_{warm} := undef	H := H′
⋮		w&r:update DIR′ w&w′: assign to V_{warm} updates: assign to $V_{H'}$	
$+3\Delta$	$V_H := V_{cool}$ discard V_{cool} and DIR	receive DIR DIR := DIR $\cup$ DIR′ if V_{warm} is defined then $V_{H'} := V_{warm}$ discard V_{warm}	

Figure 8. Moving the Hot Copy

EP $< t_r+3\Delta$. An operation with *EP* $< t_r+3\Delta$ is executed as if the migration had not begun, i.e., it is executed as in the static version with H as the owner.

EP $> t_r+4\Delta$. An operation with *EP* $> t_r+4\Delta$ is executed as if the migration were complete, i.e., it is executed as in the static version with H′ as the owner.

$t_r+3\Delta <$ *EP* $< t_r+4\Delta$. An operation with *EP* in the Δ pulses after $t_r+3\Delta$, called a *transitional* operation, is executed as in the static version with H as the owner, but is also executed locally at H′ at the same logical time as it is executed at H.

The leftmost group of operations in Fig. 9, (labeled 1-4) have *EP* $< t_r+3\Delta$, the rightmost group (9-11) have *EP* $> t_r+4\Delta$, and the middle group (5-8) are transitional operations. As in the static version, r and w operations are sent to the owner by the home MM, w′ operations are sent to the owner directly by the source PE, and r′ operations are not sent to the owner. All r, w, and w′ operations with *EP* $< t_r+3\Delta$ are sent to the owner (H) before $t_r+\Delta$, when the sender receives the COA. Hence r, w, and w′ operations with *EP* $< t_r+3\Delta$ are sent only to H. Similarly, r, w, and w′ operations with *EP* $> t_r+4\Delta$ are sent to the owner (H′) after $t_r+2\Delta$, when the sender replaces H with H′ as the owner of V. Hence r, w, and w′ operations with *EP* $> t_r+4\Delta$ are sent only to H′. Transitional operations are sent to the owner in the Δ pulses after the sender receives the COA and are sent to H′ as well as H.

Transitional operations are executed in the interval between $t_r+2\Delta$ and $t_r+3\Delta$. Transitional operations are executed at both H and H′. At H transitional operations are executed as before t_r (except in the cache updates and r responses it sends, H identifies H′ as the owner instead of itself). For example, H responds to the r operation labeled 5 by sending a response to the source PE and adding the source PE to *DIR*. At H′, by contrast, operations are only executed locally. For example, for the r operation labeled 5, H′ adds the source PE to *DIR′* but does not send a response. Note that until it executes the first transitional write (the w labeled 6), H′ does not know the value of the hot copy, V_H. The value of V_{warm} is undefined and $V_{H'}$ is a cold copy. After H′ executes the first transitional write, V_{warm} equals V_H, but H′ does not yet have sufficient information to become the owner of V because it still lacks a complete

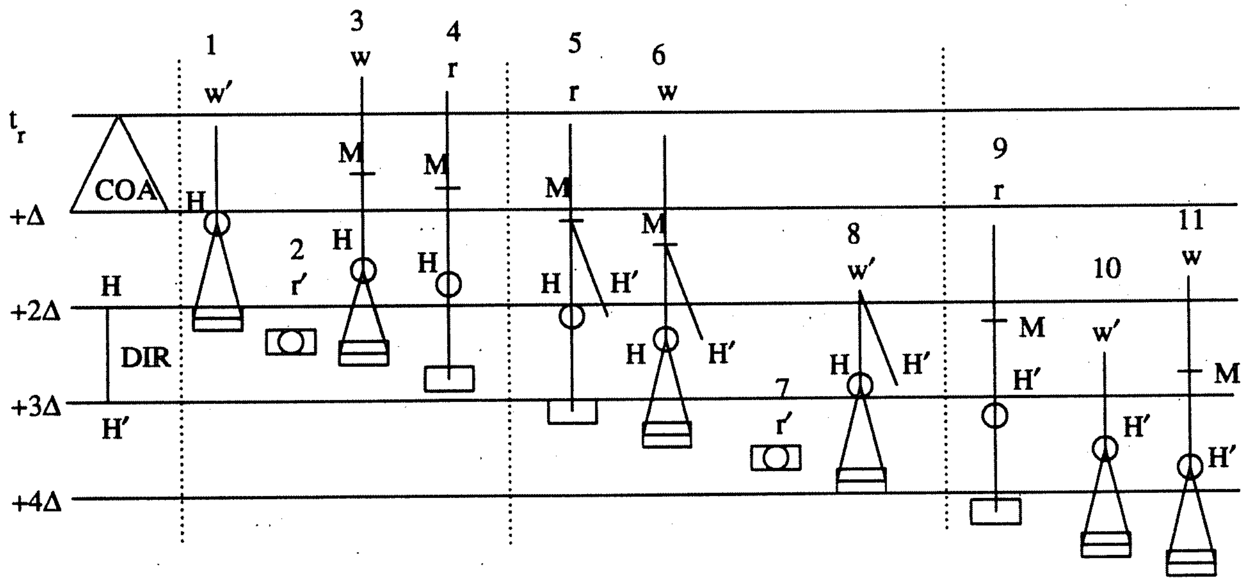

Figure 9. Writes and Reads Concurrent with Migration

directory for v. A directory is complete if it accurately reflects the set of PE's with cache copies, taking into account the cache propagation time. In this protocol, for which the update propagation time is Δ pulses, *DIR* for v is *complete* at time t if *DIR* contains the *pId*'s of exactly those PE's that have a cold copy of v at time $t+\Delta$. Hence *DIR* for v is complete if for each PE, *DIR* contains the *pId* of the PE *iff* a directory modifying operation (r, w, or *release*) on v issued by the PE is executed before t and the last such directory modifying operation to be executed before t is not a *release*.

At $t_r+3\Delta$, when it receives the copy of *DIR* sent by H Δ pulses earlier, H′ can compute the complete directory for v. Since H sent *DIR* at $t_r+2\Delta$, before executing any transitional operations, *DIR* does not reflect directory modifying transitional operations (e.g., the operations labeled 5 and 6). However *DIR′* contains the *pId*'s for all PE's emitting transitional r and w operations. (By assumption, no transitional *release* is executed.) H′ computes the complete directory for v by taking the union of *DIR′* and the copy of *DIR* received from H.

At $t_r+3\Delta$, H′ makes $v_{H'}$ the hot copy by setting it equal to v_H, the current hot copy. H′ can compute the current value of v_H (more precisely the last value of v_H at $t_r+3\Delta$ before v_H becomes a cold copy) from v_{warm} and $v_{H'}$. If a transitional write was executed, v_{warm} is defined and v_{warm} equals v_H. For the execution shown in Fig. 9, v_{warm} and v_H at $t_r+3\Delta$ both equal the value written by the w′ labeled 8. H′ makes $v_{H'}$ the hot copy by setting it equal to v_{warm}. Otherwise, no transitional write was executed, v_{warm} is undefined, and $v_{H'}$ already equals v_H. Although $v_{H'}$ is a cold copy and lags Δ pulses behind v_H, $v_{H'}$ equals v_H because no write was executed on v_H in the last Δ pulses. If neither the w nor w′ operations labeled 6 and 8 were executed, v_{warm} would be undefined at $t_r+3\Delta$ and both v_H and $v_{H'}$ would equal the value written by the last write on v, the w labeled 3. Note that though H′ is removed from *DIR* by H at $t_r+2\Delta$, H′ continues to receive cold copy updates until $t_r+3\Delta$. In particular, it receives the update sent by H in response to the w labeled 3 and performs the update on $v_{H'}$.

At $t_r+3\Delta$, H makes v_H a cold copy by setting it equal to v_{cool}, the value of the hot copy Δ pulses earlier. In the execution shown in Fig. 9, H sets v_H equal to the value written by the w labeled 3, the last write executed at H before $t_r+2\Delta$ when it set V_{cool} equal to V_H. Because it added itself to *DIR* at $t_r+2\Delta$, H receives cold copy updates beginning at $t_r+3\Delta$. In the first Δ pulses after $t_r+3\Delta$, these updates come from itself. For example, H receives the updates sent by itself in response to the w labeled 6. Updates sent by a PE to itself either traverse the ICN or are buffered Δ pulses at the PE. Because H added itself to *DIR* before sending the copy of *DIR* to H′, H receives all cold copy updates to V sent by H′. H′ begins to emit updates at $t_r+3\Delta$ and H to receive the updates from H′ at $t_r+4\Delta$.

We have made several simplifying assumptions and now describe how these assumptions can be relaxed.

(1) H′ need not have a cold copy of V before acquiring the hot copy. If H′ does not have a cold copy, it need not receive the COA. Allocation of V_{warm} and *DIR′* can be triggered by receipt of a transitional operation and transition to ownership at $t_r+3\Delta$ by receipt of the copy of *DIR* sent by H. In the case no transitional write is executed, H′ must know the value of V_H at $t_r+2\Delta$ to correctly initialize $V_{H'}$. If H′ does not have a cold copy of V, H must send H′ a copy of V_H at $t_r+2\Delta$.

(2) Releases can be executed while the hot copy is in motion. In the static version, a release is sent to the owner and the owner executes the release by deleting the source PE from *DIR*. We define a transitional release as a release executed during the Δ pulses before $t_r+3\Delta$. Each release is executed in the dynamic version in the same way as in the static version except the source PE sends a transitional release to H′ as well as to H. H executes the release as before, by removing the source PE from *DIR*, but H′ executes the release by recording the release operation in *DIR′*. Thus *DIR′* contains not only the set of PE's to be added to *DIR* but also the set to be deleted. At $t_r+3\Delta$, when it receives the copy of *DIR* sent by H it deletes from *DIR* the PE's that issued releases recorded in *DIR′*. Note these PE's will be in *DIR* because H sends *DIR* at $t_r+2\Delta$, before it executes any transitional release. After deleting these PE's, H′ computes the new directory as before.

(3) A PE can emit w′ operations on V any time after it receives a cold copy of V. Our initial assumption that a PE waits Δ pulses after receiving a cold copy before issuing a w′ operation on the same variable provides an easy but unnecessarily restrictive way to handle the case of the PE that acquires a cold copy of V in the Δ pulses after $t_r+\Delta$. During this period, all w′ operations must be sent to both H′ and H. The difficulty presented by this case is that the PE does not receive the COA because it is not in *DIR* when the COA is emitted, and therefore has no way to determine the interval during which it must send w′ operations to both H′ and H. Our initial assumption solves this problem by imposing a delay that prevents the PE from emitting a w′ on V until after $t_r+2\Delta$, when the w′ need be sent only to H′. Eliminating the assumption requires supplying the PE sufficient information to determine the interval during which to send w′'s to both H′ and H. Accordingly, we require that H respond to w and r operations in the interval between t_r and $t_r+\Delta$, with the *pld*'s of both itself and H′ and the integer $r = \Delta - (\textit{pulses elapsed since } t_r)$. Any PE acquiring a cold copy in the Δ pulses after $t_r+\Delta$ receives with its new cold copy the integer r representing the number of pulses until $t_r+2\Delta$. For the next r pulses, the PE sends any w′ to both H′ and H. Thereafter the PE sends operations to H′ only.

(4) H′ and H can emit operations on V while ownership of V is changing. H emits operations on V as w″ and r″ operations before $t_r+3\Delta$ (with one exception) and as w′ and r′ operations thereafter. Similarly, H′ emits w′ and r′ before $t_r+3\Delta$ (again with one exception) and emits w″ and r″, thereafter. The first exception is that H cannot emit w″ operations in the Δ pulses before $t_r+3\Delta$. The

reason for this restriction is that all transitional writes must be executed at the same logical time at both H′ and H. The value written by a transitional w″ cannot get to H′ until Δ pulses after it is executed at H. Fig. 10 shows the execution of operations emitted by H and H′ while ownership is changing. The w″ operation labeled 4 is illegal. Although in the Δ pulses before t_r+3Δ H cannot emit w″ operations on V, it can emit w′ operations (e.g., the w′ labeled 5 in the figure). Any such w′ operation is sent to H′ for execution, as with any w′ emitted in the same interval. The second exception is optional. At any time after it executes a transitional write, H′ has the option of reading V using an r″ instead of an r′. Any such r″ is executed on V_{warm}. After execution of a transitional write V_{warm} equals V_H, the hot copy.

(5) H need not keep a cold copy of V when it relinquishes the hot copy. If H does not need to retain a cold copy of V, it can omit the steps at t_r+2Δ of adding itself to *DIR* and allocating V_{cool} and at t_r+3Δ of making V_H a cold copy and discarding V_{cool}.

Ownership can change every 3Δ pulses. A PE must own V before it can relinquish the hot copy of V. We may consider relaxing this restriction, requiring instead that a PE must have received a COA naming it as the next owner and know the successor to itself is on the request queue before it can emit a COA. This lesser restriction would allow ownership to change every Δ pulses.

The space-time diagram in Fig. 11 shows the relationship between the copies of V while the hot copy of V migrates from H to H′. Each vertical dotted line represents a logical time. The horizontal dotted lines represent copies of V at different locations: the top line represents V_H; the bottom line $V_{H'}$; and

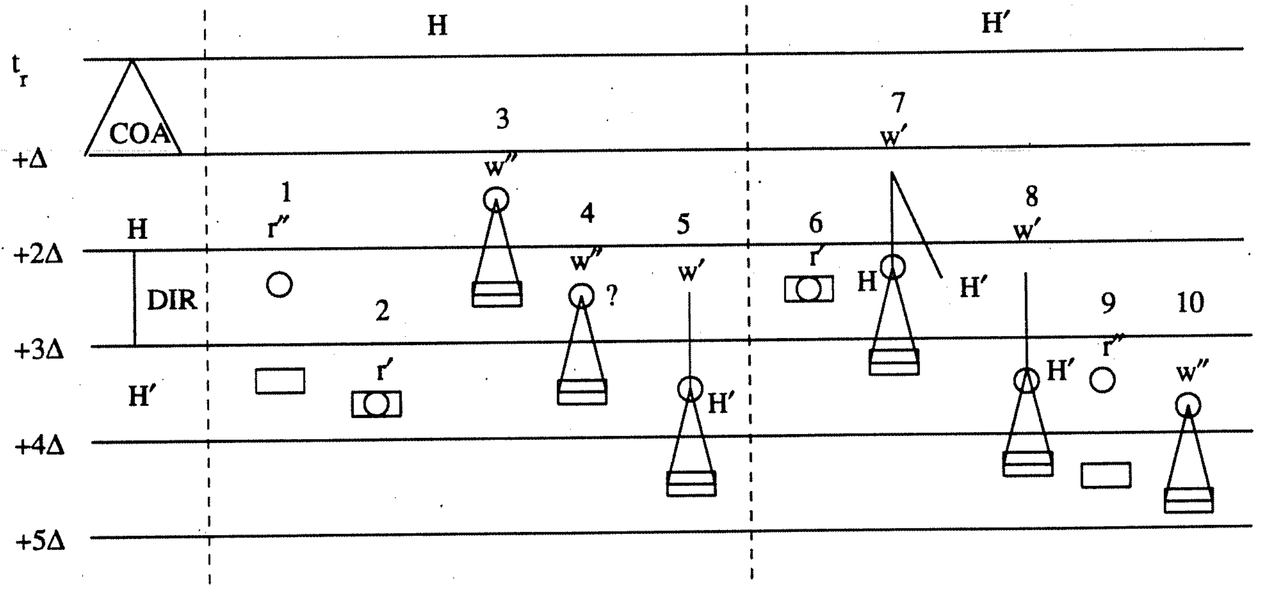

Figure 10. Operations by H and H′ Concurrent with Migration

the middle line, the memory copy and any other cold copies of V. The solid lines are iso-value lines, i.e., each line connects points of equal value. For example, V_H at t_r equals $V_{H'}$, the memory copy, and any other cold copy Δ pulses later. The arcs represent cold copy updates sent by H to itself. Before t_r+3Δ the value of V_H at time t equals the value of the memory copy and all cold copies at time t+Δ. After t_r+3Δ the value of H′ at time t equals the value of the memory copy and all cold copies at time t+Δ. Thus the relationships between cache values shown in the figure conform to the δ-invariant.

We show the *EP* for each operation is independent of whether ownership is changing. Moving the hot copy does not affect the length of the interval between the time a write is emitted and the time it is executed on the memory copy. Thus moving the hot copy does not affect the *EP* for a write. Similarly, moving the hot copy does not affect the length of the interval between the time a read is emitted and the time it is executed on the hot copy (in the case of an r or r″) or on the local cold cache (r′). Assuming the δ values are the same as in the static version, 0 for a cold copy and Δ for the hot copy, the *EP* for reads is the same as in the static version. Thus, assuming the δ-invariant holds, moving the hot copy does not affect the *EP* for a read. A proof that the migration algorithm maintains the δ-invariant is in the appendix to this paper. Because the *EP*'s for operations are not affected by migration, PE-SIU's can maintain

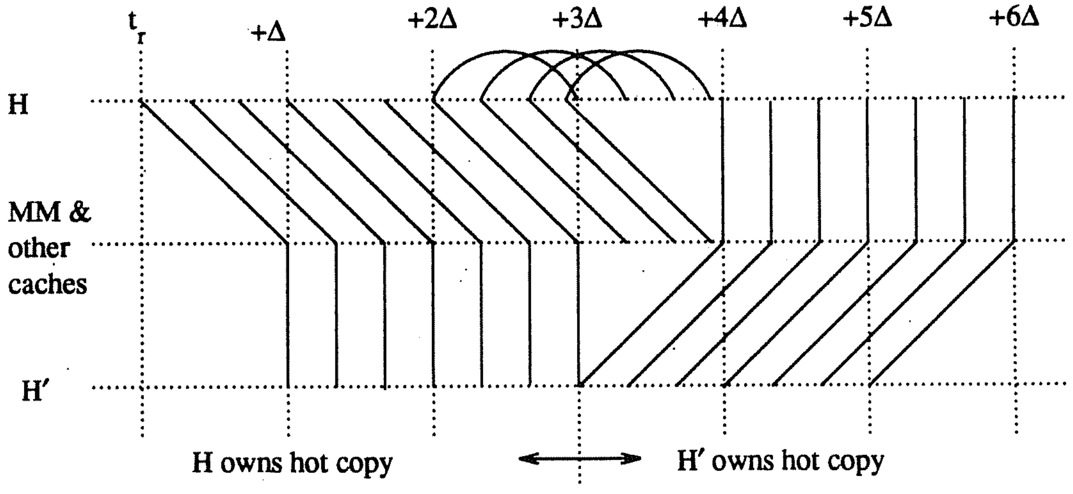

Figure 11. δ-Invariant During Hot Copy Migration

memory coherence for isochron programs by following the emission rules as in the static version of the early cache protocol.

4.3. Atomicity

The emission rules ensure atomic execution of isochrons, but isochrons represent only a limited class of atomic actions. Operations in an isochron must be issued as a batch, so operations with data dependencies cannot be executed in the same isochron. We have proposed techniques based on isochrons together with *access sequences* and *split operations*, defined below, to support a broad class of atomic actions [WiR89]. The techniques extend to systems with caches managed by delta-cache protocols. A major benefit of the techniques is that they do not require operations on locks. The techniques described in this section are alternatives to the technique discussed in the section on the early cache protocol of executing atomic actions by acquiring ownership of the accessed variables.

An access sequence for a variable is the sequence of accesses made to the variable over time. Each element in the access sequence represents an access to the variable and either records the value read or written by the access or reserves a position for the access. Split operations are the set of operations defined for the access sequence representation and are based on the idea of splitting an access into two steps — a scheduling step that appends an element to the access sequence to reserve the context for an access and an assignment step that transfers a value. For writes, the transfer is from a process's local variable or register to the element appended by the scheduling step for the write. For reads, the direction of transfer is reversed, from the access sequence to the local variable or register. Splitting a write into two steps allows the write to be scheduled before the value to be written is known. The steps can be collapsed into a single step when the process already knows the value to be written when it schedules the write. An MM schedules a write by appending an element with the special value nil denoted "Λ" and returning the identifier TAG of this element. When it determines the value to be written, the process sends both the value and the TAG returned by the scheduling step. The TAG enables the MM to assign the value to the element reserved for it. For each read, the MM returns the value assigned by the preceding write, i.e., the write appending the element most closely preceding the read in the access sequence. If the value of the element appended by the preceding write is Λ, the MM executes the assignment step for the read as part of the assignment step for that write.

A process executes an atomic action by issuing an isochron that schedules all the accesses required for the atomic action, executing the assignment steps for these accesses as it determines the values to be assigned. Execution is atomic because the isochron used to schedule the accesses reserves a consistent "time slice" across the histories of the accessed variables. This technique, called the *scheduling isochron* technique, works for atomic actions with access sets that can be determined at the beginning of execution of the atomic action. We have proposed variations on the technique for atomic actions with data dependent access sets [WiR89]. Both the scheduling isochron technique and the techniques for executing data dependent atomic actions extend to systems with caches. We describe here how to use the scheduling isochron technique with the late cache protocol. The technique can be similarly extended to other delta-cache protocols.

The home MM for a variable v maintains the current value of v and a version identifier TAG. When it executes an operation scheduling a write, the MM assigns the value Λ to the memory copy, assigns a new version identifier TAG, and sends both the value Λ and the TAG to all processes with cache copies, i.e., the scheduling operation is treated as a write storing the value Λ. The MM executes the assignment step for a write by multicasting cache updates giving the value assigned and the TAG supplied with the assignment by the source PE. The value in memory changes only if the supplied TAG matches the current TAG for v. The MM executes an r by sending the current TAG and value, possibly Λ, to the source PE.

The value returned by an r on v is used as before to initialize a new cache copy of v. A cache copy records both a value and a TAG. If the value returned by an r is Λ, the cache records the value Λ and TAG and the destination address for the unsatisfied read, returning a value for the r only when the write is substantiated. The cache recognizes the cache update corresponding to the unsubstantiated write by the TAG supplied with the cache update. An r' executed when the value of the cache copy is Λ is executed similarly. If a process may have multiple outstanding reads for the same variable, the cache may have multiple cache copies of the same variable. If reads are blocking, the cache need contain at most two copies of any given variable: the first representing a pending write that must be substantiated to satisfy a locally issued read, and the second the current value of the variable. The second copy can be eliminated if the home MM sends the current value and TAG with each cache update.

With the introduction of caches, the need for access sequences disappears. More precisely, the representation of access sequences becomes distributed, each element becoming a copy in cache, where it satisfies the additional, independent goal of providing a local copy to access. Without caches, the scheduling isochron technique requires that each MM store access sequences recording the reads scheduled after each write so that the MM can identify the processes to which to send the new value when a write is substantiated. With caches, each write is sent to all of the processes with a cache copy. Thus at the cost of informing more processes than may be necessary, the MM need maintain for each variable only the directory giving the location of cache copies and the TAG and value of the last write access.

5. CONCLUSION

This paper is an exploration of the idea of combining synchronicity and asynchronicity to secure the benefits of both. We have demonstrated that asynchronous systems can use low-level synchrony within an ICN as the basis for concurrency control and memory coherence.

We have proposed a new correctness criterion for cache coherence protocols and described a family of new cache coherence protocols called δ-cache protocols based on the isotach network. The disadvantage of the δ-cache protocols is the additional cost of isotach networks in communication latency and hardware. The advantage is that the δ-cache protocols are more highly concurrent than other directory protocols in that they permit more pipelining, allow multiple readers and writers to the same cache block, and are compatible with techniques for executing atomic actions accessing multiple shared variables without acquiring exclusive access rights to the variables.

Our current and future work includes evaluating the δ-protocols, exploring the use of compiler analysis in choosing the most efficient delta-cache protocol to use for each cache block, and extending the protocols to the related problem of implementing a virtual shared memory on a distributed memory machine. We are continuing our study of memory coherence based on isotach networks by working toward the following types of results:

(1) Performance evaluation. The efficiency of the delta-cache protocols depends on the efficiency of the isotach network. The performance of isotach networks is the subject of current work. In parallel with a performance study of the isotach network, we may begin a performance study of the delta-cache protocols using a workload model similar to the model proposed by Dubois and Briggs [DuB82], leaving the cost of communication as a parameter. The workload model has been used as the basis for several analytic and simulation studies of multiprocessor caches [ArB84, ArB86, DuW88, LeR90, MBK90]. We prefer this method to trace simulation because trace simulation assumes reference patterns captured in the trace are representative of those that would be generated on a system using the protocol being tested. In evaluating the delta-cache protocols, this assumption is more questionable than in studies of other cache protocols because both the concurrency control mechanism and the cache protocol differ from the system generating the trace. In particular, we do not want to test the performance of delta-cache protocols using a trace containing lock operations, since our protocols do not require lock operations.

(2) A more detailed description of the delta-cache protocols addressing the issues of cache organization, replacement policy, block size, mechanism for handling process migration and processor multitasking, address translation method, and directory representation.

(3) More delta-cache protocols. We intend to continue to explore the design space for cache protocols defined in this paper by describing on-time protocols, and protocols for network topologies in which a PE can be directly connected to an MM such as the BBN Butterfly. Assuming reasonable deadlock-free routing algorithms can be found for hypercubes and meshes, we intend to describe

protocols for these topologies.

(4) Software hybrids. Hybrid hardware/software cache protocols take advantage of both static and run-time information. We may investigate hybrid protocols that use compiler analysis to select among delta-cache protocols on a per block or per reference basis.

(5) Adaptive protocols. The algorithm for changing the owner of a variable described in the early cache protocol migrates the hot copy in response to explicit requests for ownership added to the program by the programmer or compiler. We intend to consider an alternative migration algorithm that migrates the hot copy adaptively, in response to the pattern of references to the variable. We also intend to consider how to change the cache coherence protocol dynamically in response to reference patterns, e.g. changing the cache protocol for v from the early to the late cache protocol in response to high contention for v.

(6) Techniques to enforce version consistency compatible with the delta-cache protocols. An execution of a parallel program is version consistent if it respects the data dependencies implied by the program, i.e., if accesses to the same variable by different processes are executed in the order specified by the program where an order is specified. Version consistency is typically enforced by semaphores and barriers. For systems without caches, we have proposed the *synch* operation as an alternative mechanism for enforcing version consistency [WiR89]. A *synch* operation is a split operation scheduling an access in which the assignment step is executed by another process. Synch operations are a wait-free mechanism for enforcing write/write and read/write data dependencies and are compatible with the techniques we have proposed for enforcing sequential consistency and atomicity. We intend to extend this technique to systems with caches.

(7) Combining. Delta-cache protocols do not require a combining network, but may be compatible with combining. In a combining network, operations assessing the same variable are combined when they collide at switches in the ICN, reducing traffic and serialization of access at the memory module level [KRS88]. We have shown isochrons containing only read and write operations are

combinable [RWW89]. We intend to determine whether isochrons containing split operations are combinable and whether combining can extend to systems with caches. Combining could be used in systems with caches to avoid serial execution of multiple concurrent writes to the same variable and to reduce traffic in response to multiple requests for cache copies of the same block.

(8) Extension to secondary memory. By considering systems with caches, we have begun to extend concurrency control based on local synchrony into the memory hierarchy. We intend to consider how to incorporate secondary memory as well as main memory and caches.

(9) Application of protocols to distributed systems. We intend to explore other applications of this work, in particular to virtual or distributed shared memory [Li89]. In studying cache coherence, we are exploring ways to increase the availability of data by migration and replication while maintaining serializability. The same problem arises in message-based programming on distributed memory multiprocessors and in geographically distributed systems. Techniques we devise in studying cache coherence may apply in these other areas.

REFERENCES

[AdH90] S. V. Adve and M. D. Hill, Weak Ordering --- A New Definition, *Proc. of the 17th International Symp. Computer Architecture*, 1990, 2-11.

[Aga88] A. Agarwal, et al., An Evaluation of Directory Schemes for Cache Coherence, *Proc. of the 15th International Symp. Computer Architecture*, 1988, 280-289.

[ArB84] J. Archibald and J. L. Baer, An Economical Solution to the Cache Coherence Problem, *Proc. 11th International Symp. Computer Architecture*, 1984, 355-362.

[ArB86] J. Archibald and J. L. Baer, Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model, *ACM Transactions on Computer Systems 4*,4 (November 1986), 273-298.

[Awe85] B. Awerbuch, Complexity of Network Synchronization, *J. ACM 32*,4 (October 1985), 804-423.

[BaR89] S. J. Baylor and B. D. Rathi, A Study of the Memory Reference Behavior of Engineering/Scientific Applications in Parallel Processors, *ICPP*, 1989, I-78-82.

[CeF78] L. M. Censier and P. Feautrier, A New Solution to Coherence Problems in Multicache Systems, *IEEE Trans. on Computers*, December 1978, 1112-1118.

[ChV88] H. Cheong and A. Veidenbaum, Proc. 15th International Symp. Computer Architecture, 1988.

[DuB82] M. Dubois and F. A. Briggs, Effects of Cache Coherency in Multiprocessors, *IEEE Trans. on Computers 31*(November 1982), 1083-1099.

[DSB86] M. Dubois, C. Scheurich and F. Briggs, Memory Access Buffering in Multiprocessors, *Proc. 13th International Symp. Computer Architecture*, 1986, 434-442.

[DuW88] M. Dubois and J. Wang, Shared Data Contention in a Cache Coherence Protocol, *ICPP 1988*, 1988.

[EgK88] S. Eggers and R. Katz, A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation, *Proc. 15th International Symp. Computer Architecture*, May 1988, 373-382.

[Gib89] P. B. Gibbons, The Asynchronous PRAM: A Semi-Synchronous Model for Shared Memory MIMD Machines, Tech. Rep.-89-062, International Computer Science Institute, Berkeley, California, December, 1989.

[KRS88] C. P. Kruskal, L. Rudolph and M. Snir, Efficient Synchronization on Multicomputers with Shared Memory, *Trans. Prog. Lang and Systems 10*,4 (October 1988), 579-601.

[Lam78] L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, *Comm. of the ACM 21*,7 (July 1978), 558-565.

[Lam79] L. Lamport, How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs, *IEEE Trans. on Computers 28*(1979), 690-691.

[LeR90] J. Lee and U. Ramachandran, Synchronization with Multiprocessor Caches, *Proc. 17th International Symp. Computer Architecture*, 1990, 27-37.

[Li89] K. Li and P. Hudak, Memory Coherency in Shared Virtual Memory Systems, *ACM Trans. Computer Systems 7*,4 (November 1989), 321-359.

[Mat88] F. Mattern, Virtual Time and Global States of Distributed Systems, *Parallel and Distributed Algorithms*, 1988, 215-226.

[MiB89] S. L. Min and J. Baer, A Timestamp-based Cache Coherence Scheme, *Int. Conf. on Parallel Processing 1*(1989), 23-32.

[MBK90] S. L. Min, J. Baer and H. Kim, An Efficient Caching Support for Critical Sections in Large-Scale Shared-Memory Multiprocessors, RC 15311, IBM Research Report, January 1990.

[Pap86] C. Papadimitriou, *Database Concurrency Control*, Computer Science Press, 1986.

[Ran87] A. G. Ranade, How to Emulate Shared Memory, *IEEE Annual Symp. on Foundations of Computer Science*, Los Angeles, 1987, 185-194.

[RBJ88] A. G. Ranade, S. N. Bhatt and S. L. Johnson, The Fluent Abstract Machine, Tech. Rep. 573, Yale University, Dpt. of Computer Science, January, 1988.

[RWW89] P. F. Reynolds, Jr., C. Williams and R. R. Wagner, Jr., Parallel Operations, Tech. Rep. 89-16, University of Virginia, Department of Computer Science, December, 1989.

[ScD87] C. Scheurich and M. Dubois, Correct Memory Operation of Cache-Based Multiprocessors, *Proc. 14th Int. Symp. Computer Architecture*, June 1987, 234-243.

[Ste89] P. Stenstrom, A Cache Consistency Protocol for Multiprocessors with Multistage Networks, *Proc. 16th International Symp. Computer Architecture*, May 1989, 407-415.

[WiR89] C. Williams and P. F. Reynolds, Jr., On Variables as Access Sequences in Parallel Asynchronous Computations, Tech. Rep. 89-17, University of Virginia, Department of Computer Science, December, 1989.

APPENDIX: Maintaining the δ-Invariant During Migration

We show the algorithm for changing the owner of the hot copy in the early cache protocol described in section 4.2.2 maintains the δ-invariant by showing the invariant holds before, after, and during the migration of the hot copy from the old owner H to the new owner H′. The δ-invariant

$$V_{hot,i} = V_{memory,i+\Delta} = V_{cold,i+\Delta}$$

means the hot cache copy of V at time (i,j,k) equals the memory and cold copies at time $(i+\Delta,j,k)$. Let t_w be the execution time of the first write executed after $t_r+3\Delta$. We show the δ-invariant holds for all logical times t: (1) $t < t_r+3\Delta$; (2) $t_r+3\Delta \le t < t_w$; and (3) $t_w \le t$. The relationships among the hot, memory, and cold copies of V remains the same, though the location of the hot copy changes from H to H′ at $t_r+3\Delta$.

(1) $t < t_r+3\Delta$. Before $t_r+3\Delta$, the δ-invariant holds with V_H the hot copy. Each write to V executed before $t_r+3\Delta$ is executed at H on V_H. For each write it executes before $t_r+3\Delta$, H sends updates to the home MM and each PE ∈ *DIR*. Each update is received exactly Δ pulses after it is sent. Hence for all t, $t < t_r+3\Delta$, the hot copy, V_H, at t equals the memory copy at $t+\Delta$. In considering the relationship between the hot and cold copies we first consider the cold copies at H′ and H.

H′ has a cold copy, $V_{H'}$, until $t_r+3\Delta$, when $V_{H'}$ becomes the hot copy. For all t, $t < t_r+2\Delta$, V_H at $t = V_{H'}$ at $t+\Delta$ because H′ ∈ *DIR* until $t_r+2\Delta$. H continues to send cache updates for V to H′ until $t_r+2\Delta$ so H′ continues to receive updates until $t_r+3\Delta$. Though H′ receives both cache updates and write operations for V in the Δ pulses before $t_r+3\Delta$, it executes the write operations only on V_{warm} and performs the cache updates only on $V_{H'}$. Since $V_{H'}$ ceases to be a cold copy at $t_r+3\Delta$, its value after $t_r+3\Delta$ is not constrained by the δ-invariant to be the value of V_H Δ pulses earlier.

H has a cold copy of V only after $t_r+3\Delta$, when V_H ceases to be the hot copy. Because V_H is the hot copy before and a cold copy after $t_r+3\Delta$, the δ-invariant implies for all times t, $t_r+2\Delta \le t < t_r+3\Delta$, V_H at $t = V_H$ at $t+\Delta$. This relationship holds for $t = t_r+2\Delta$ because H assigns the value of V_{cool} to V_H at $t_r+3\Delta$ and V_{cool} is the value of V_H at $t_r+2\Delta$. It continues to hold for t, $t_r+2\Delta \le t < t_r+3\Delta$, because V_H adds itself to *DIR* at $t_r+2\Delta$. For each write executed by H in the Δ pulses before $t_r+3\Delta$, H sends an update to itself that is received and executed exactly Δ pulses later.

The δ-invariant holds for the remaining cold copies, the copies of $\mathtt{v}$ at PE's other than $\mathtt{H}$ and $\mathtt{H}'$ because *DIR* is *complete*. The *DIR* at $\mathtt{H}$ is complete through t_r+3Δ because all $\mathtt{w}$, $\mathtt{r}$, and release operations executed on $\mathtt{v}$ until t_r+3Δ are executed at $\mathtt{H}$. Because *DIR* is complete and $\mathtt{H}$ sends an update to each PE $\in$ *DIR* for each write executed on $\mathtt{v}$ before t_r+3Δ, the value of $\mathtt{v}_{\mathtt{H}}$ at t, $t<t_r+3\Delta$, equals the the value of each cold copy of $\mathtt{v}$ at $t+\Delta$.

(2) $t_r+3\Delta \le t < t_w$. From t_r+3Δ until the first write is executed after t_r+3Δ, the δ-invariant holds with $\mathtt{v}_{\mathtt{H}'}$ the hot copy. At t_r+3Δ, the values of $\mathtt{v}_{\mathtt{H}'}$ and $\mathtt{v}_{\mathtt{H}}$ are swapped, i.e., $\mathtt{H}'$ assigns to $\mathtt{v}_{\mathtt{H}'}$ the value of $\mathtt{v}_{\mathtt{H}}$ immediately before t_r+3Δ, and $\mathtt{H}$ assigns to $\mathtt{v}_{\mathtt{H}}$ the value of $\mathtt{v}_{\mathtt{H}'}$ immediately before t_r+3Δ. Since $\mathtt{H}$ is the hot copy and $\mathtt{H}'$ a cold copy before t_r+3Δ, we show $\mathtt{H}'$ is the hot copy and $\mathtt{H}$ a cold copy at t_r+3Δ by showing the swap occurs as stated.

The swap occurs atomically because the assignments to $\mathtt{H}'$ and $\mathtt{H}$ occur at the same logical time t_r+3Δ and all other accesses to $\mathtt{v}$ are executed either before or after t_r+3Δ. No other access is executed at t_r+3Δ because no other access executed in the same pulse as the swap has both the same *tick* and *tock* components as the COA.

At t_r+3Δ, $\mathtt{H}'$ assigns $\mathtt{v}_{\mathtt{H}'}$ the value of $\mathtt{v}_{\mathtt{H}}$ immediately before t_r+3Δ, i.e., the value of $\mathtt{v}_{\mathtt{H}}$ before the swap. There are two cases. If $\mathtt{H}'$ executed one or more writes during the interval between t_r+2Δ and t_r+3Δ (transitional writes) $\mathtt{v}_{\mathtt{warm}}$ at t_r+3Δ equals $\mathtt{v}_{\mathtt{H}}$. Since each transitional write is executed at the same logical time on both $\mathtt{v}_{\mathtt{warm}}$ and $\mathtt{v}_{\mathtt{H}}$, and at least one transitional write is executed, $\mathtt{v}_{\mathtt{warm}}$ at $t_r+3\Delta = \mathtt{v}_{\mathtt{H}}$ at t_r+3Δ. In the latter case, no transitional write is executed. Since no transitional write is executed, $\mathtt{v}_{\mathtt{H}}$ at $t_r+3\Delta = \mathtt{v}_{\mathtt{H}}$ at t_r+2Δ. Since $\mathtt{v}_{\mathtt{H}'}$ is a cold copy until t_r+3Δ, $\mathtt{v}_{\mathtt{H}'}$ at $t_r+3\Delta = \mathtt{v}_{\mathtt{H}}$ at t_r+2Δ. Thus $\mathtt{v}_{\mathtt{H}'}$ at $t_r+3\Delta = \mathtt{v}_{\mathtt{H}}$ at t_r+3Δ. At t_r+3Δ, $\mathtt{v}_{\mathtt{H}'}$ is assigned the value of $\mathtt{v}_{\mathtt{warm}}$ in the first case, and is assigned no new value in the latter case. In both cases, $\mathtt{v}_{\mathtt{H}'}$ at $t_r+3\Delta = \mathtt{v}_{\mathtt{H}}$ at t_r+3Δ.

We now consider part of the swap occurring at $\mathtt{H}$. At t_r+3Δ, $\mathtt{H}$ assigns $\mathtt{v}_{\mathtt{H}}$ the value of $\mathtt{v}_{\mathtt{H}'}$ immediately before t_r+3Δ, i.e., the value of $\mathtt{v}_{\mathtt{H}'}$ before the swap. The value assigned to $\mathtt{v}_{\mathtt{H}}$ at t_r+3Δ is

$\mathtt{v}_{\mathtt{cool}}$, the value of $\mathtt{v}_{\mathrm{H}}$ at $t_r+2\Delta$. Because $\mathtt{v}_{\mathrm{H}'}$ is a cold copy, lagging Δ pulses behind the hot copy, the value assigned to $\mathtt{v}_{\mathrm{H}}$ also equals the value of $\mathtt{v}_{\mathrm{H}'}$ at $t_r+3\Delta$ before the swap.

The invariant holds for all t, $t_r+3\Delta < t < t_w$ because it holds at $t_r+3\Delta$ and the value of all copies of $\mathtt{v}$ remains constant during the relevant period: for the hot copy, the time from $t_r+3\Delta$ until t_w; for all other copies, the time from $t_r+4\Delta$ until $t_w+\Delta$. By definition of t_w, H′ does not change $\mathtt{v}_{\mathrm{H}'}$ or send any cache updates for $\mathtt{v}$ from $t_r+3\Delta$ until t_w. Since all cache updates for $\mathtt{v}$ received after $t_r+4\Delta$ are sent by H′, the home MM and PE's with cold copies of $\mathtt{v}$ can receive no cache update for $\mathtt{v}$ between $t_r+4\Delta$ and $t_w+\Delta$. Hence the memory and cold copies remain constant during this period.

(3) $t_w \le t$. After the first write to $\mathtt{v}$ executed after $t_r+3\Delta$, the δ-invariant holds with $\mathtt{v}_{\mathrm{H}'}$ as the hot copy. Each write executed after $t_r+3\Delta$ is executed at H′ on $\mathtt{v}_{\mathrm{H}'}$. For each write executed by H′ after $t_r+3\Delta$, H′ sends updates received at the home MM and each PE $\in$ *DIR* Δ pulses later. Hence, for t, $t_w \le t$, $\mathtt{v}_{\mathrm{H}'}$ at t equals the value of the memory copy at $t+\Delta$. We next consider the relationship between $\mathtt{v}_{\mathrm{H}'}$ and the cold copies. H′ does not have a cold copy of $\mathtt{v}$ after $t_r+3\Delta$. The invariant holds for the cold copy at H because H inserts its own *pId* into *DIR* before sending a copy of *DIR* to H′ at $t_r+2\Delta$. Hence H′ sends an update to H for each write it executes after $t_r+3\Delta$ and the value of $\mathtt{v}_{\mathrm{H}'}$ at $t, t \ge t_w$, is the value of $\mathtt{v}_{\mathrm{H}}$ at $t+\Delta$.

The invariant holds for cold copies at PE's other than H′ and H because the copy of *DIR* (for $\mathtt{v}$) at H′ is complete. We have shown *DIR* at H is complete at $t_r+3\Delta$. We show *DIR* at H′ at $t_r+3\Delta$ is complete by showing it is identical to *DIR* at H at $t_r+3\Delta$. There are two cases. First, if no transitional `r`, `w`, or *release* is executed, H′ sets *DIR* equal to the copy of *DIR* sent by H at $t_r+2\Delta$ and received at $t_r+3\Delta$. Since transitional operations are executed at both H and H′, the absence of transitional `r`, `w`, and release operations means H makes no changes to *DIR* during the Δ pulses before $t_r+3\Delta$. Hence the copy of *DIR* H′ receives at $t_r+3\Delta$ equals *DIR* at H at $t_r+3\Delta$. Otherwise, H changes *DIR* during the Δ pulses before $t_r+3\Delta$. Since H′ records in *DIR′* the *pId* of the source PE for every `r`, `w`, and *release* executed by H in the Δ pulses before $t_r+3\Delta$, *DIR′* contains a record of each change H makes to *DIR* during this interval. At $t_r+3\Delta$, when it receives the copy of *DIR* sent by H at $t_r+2\Delta$, H′ can compute *DIR* at H at $t_r+3\Delta$ from the copy of

DIR it receives together with *DIR'*. After t_r+3Δ, *DIR* at H′ remains complete because all w, r, and *release* operations on V executed after t_r+3Δ are executed at H′.