A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus

Paul Sweazey* and Alan Jay Smith**†

Abstract

Standardization of a high performance blackplane bus, so that it can accommodate boards developed by different vendors, implies the need for a standardized cache consistency protocol. In this paper we define a class of compatible consistency protocols supported by the current IEEE Futurebus design. We refer to this class as the MOESI class of protocols; the term "MOESI" is derived from the names of the states. This class of protocols has the property that any system component can select (dynamically) any action permitted by any protocol in the class, and be assured that consistency is maintained throughout the system. Included in this class are actions suitable for copyback caches, write through caches and non-caching processors. We show that the Berkeley protocol and the Dragon protocol fall within this class, and can be extended to be compatible with other members of the class. The Illinois, Firefly and Write-Once protocols can be adapted to be compatible with this class; the facilities of the Futurebus currently do not support those protocols without adaptation. We discuss very briefly performance choices among protocols, and also other issues relating to a standard bus consistency protocol.

A thought: "*A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and devines.*" [Emer41]

1. Introduction

Computer systems based on a common (shared) backplane bus have become popular for a number of reasons: (a) The cost of such a system tends to be low. (b) The backplane bus provides a standard interconnect which only has to be designed once. (c) The standard interconnect permits independent design of individual boards. (d) It also permits those independently designed boards to come from different vendors (mix and match). (e) Additional boards can be added to such a system, to incrementally add features such as I/O processors, memory or additional CPUs. The advantages of such a design have led to a number of standard buses, including the S-100, VME bus [Fish84, P1014], Multibus [Coop83], Multibus-II, Nu Bus [Texa83], IEEE Fastbus [ANSI86], et cetera; see [Gust84, Borr85] for an overview. Each new bus has been introduced to remedy real or perceived deficiencies in then available buses.

Existing buses are not considered by many to be adequate to support the new generation of high performance 32-bit microprocessors such as the Fairchild Clipper [Cho86, Fair85], The MIPS machine [Mous86], the 68020 [Moto84], the Z80000 [Phil85], the 80386 [Elay85], and the 32332 [Mate85]. A bus for this class of processor should be able to provide very high bandwidth for block transfers, while still permitting slower and less costly system components to communicate at lower rates. The need for a bus with these characteristics prompted the IEEE to set up the Futurebus (P896) standards committee, with a charter to propose a standard design for a high performance bus. The signal lines and electrical characteristics of Futurebus have been largely decided [Borr84, Bala84, P896] and are close to formal adoption.

An important application for a backplane bus is to support multi-(micro)-processor systems. The trend of technology over the last few years has led to the current situation in which it is considerably less expensive to provide N mips via K processors of N/K mips each than to build a single (uniprocessor) which delivers N mips [Smit84b]. Multiprocessor systems built around a backplane bus not only provide a convenient way to configure such systems but also provide expansibility - the number of processors can be increased as more processing power is needed. (We note, but do not further discuss, the fact that successfully programming such systems is difficult.)

Two factors require that high performance multiprocessor systems have cache memories [Smit82, 84a]. First, the access time to main memory across a bus, no matter how fast the bus and the main memory (within reason), is likely to be so large as to appreciably slow down the processor. Second, simple calculations will show that no feasible bus design can provide adequate bandwidth to memory for any reasonable number of high performance processors. A cache memory (of sufficient size and adequate performance) solves the first problem by reducing the average memory access time substantially - by as much as 90% [Smit82]. The cache also cuts the memory bandwidth requirement, since most references are satisfied locally with no bus activity. Cache memories, thus, are a necessity for high performance multiprocessor systems with a shared memory.

Cache memories, however, introduce the *cache consistency problem,* which refers to the fact that the contents of a given location of main memory can reside simultaneously in both main memory and in zero or

*Graphics Workstation Division, Tektronix, Inc., Wilsonville, Oregon 97070, USA

**Computer Science Division, EECS Department, University of California, Berkeley, Ca. 94720, USA.

†The material presented here is partially based on research supported in part by the National Science Foundation under grant DCR-8202591, by the Defense Advance Research Projects Agency (DoD), under Arpa Order No. 4871, Monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089, and by the State of California under the MICRO program.

0884-7495/86/0000/0414$01.00 © 1986 IEEE

more cache memories. If such a system is to correctly and deterministically execute computations, all references to a given location, no matter from which processor they originate, should reference the same value; i.e. the contents of the cache memories must be consistent.

There are a number of approaches to the cache consistency problem; the features and operation of the chosen solution depend largely on the type of computer architecture in which they are to operate. The cache can be shared; there can be a combination of private caches and a shared cache, with the shared cache holding the shared data; data modifications can be broadcast, with all other processors invalidating their copies; or a global or distributed directory can be maintained to keep track of which cache has what information [Arch84, Cens78, Tang76]. Surveys of the various cache consistency mechanisms appear in [Smit82, 84a, 85d], directions for memory hierarchy and cache research are discussed in [Smit85a] and a complete bibliography of the literature on cache memories appears in [Smit86].

With the increasing performance of microprocessors, a solution to the cache consistency problem in a system with buses became necessary and the first published bus consistency protocol appeared in [Good83]; such protocols are means of maintaining distributed directories. Other bus based consistency protocols have been given in [Fran84, Papa84, Rudo84, and Katz85]. An excellent survey and comparison of some of these appears in [Arch85], and another comparison appears in [Vern85].

It seems clear that any new bus design must incorporate features and facilities sufficient to support one or more of the bus based cache consistency protocols. To the extent that these protocols are of significantly different performance, support for the higher (highest) performance protocols should be available. Conversely, it is also important that the bus be able to support lower performance (i.e., lower cost) cards, and in particular, the bus must be able to support both sophisticated cache masters (i.e., copy back caches), simpler ones (e.g., write through caches) and non-caching components (e.g. I/O processors). As explained below, the IEEE Futurebus has been designed with these goals in mind.

Because the Futurebus is intended to be a standard interface to which can be attached boards from different vendors, it would seem highly desirable that the Futurebus standard specify a protocol (or a class of compatible protocols), such that any board adhering to that standard could expect to maintain cache consistency; otherwise, in general, consistency could not be ensured. For that reason, the IEEE P896.2 working group on caching in the Futurebus has been working on defining a standard protocol (or class of compatible protocols) for cache consistency. In this paper, we present **a class of compatible consistency protocols** which are supported by the IEEE Futurebus design. By **a class of protocols**, we mean that different caches/processors may use different algorithms for what to cache when. By **compatible**, we mean that as long as each cache/processor uses a protocol from the class we define, the overall memory system state will remain consistent. Our class of compatible protocols, as we explain below, includes those suitable for non-caching boards (e.g. I/O processors) and write through caches, as well as copy-back caches.

In the next section of this paper (2), we summarize, to the extent useful for our discussion, the electrical characteristics of the Futurebus. That is followed in section 3 by our MOESI model for consistency schemes, a definition of the signal lines we need, and the class of consistency protocols that we specify. In section 4, we discuss the extent to which existing protocols are compatible with the class we define. Other issues in proposing a cache consistency standard are briefly considered in section 5. Conclusions and overview appear in section 6.

2. Futurebus Electrical Features

The Futurebus incorporates numerous features that support the efficient implementation of a variety of cache consistency protocols. Some of these features were added solely to support caching, while others are a natural consequence of designing a generic, high-performance, asynchronous bus. In this section, we describe the electrical and related characteristics of the bus.

2.1. Broadcast Address Cycle

Common to all of the possible Futurebus cache consistency protocols is the need to monitor system bus activity by caches that are not currently accessing the bus; this activity is sometimes called *snooping*. Shared memory image integrity is maintained by the ability of each cache to detect when other modules are performing an action that might affect the state of a line in the local cache. This means that all caches must participate in the address-time control handshake.

On the Futurebus all address cycles are broadcast to all subsystems. A single bus master issues an address and an address strobe, and it must continue to assert the address until all other Futurebus modules signal that they no longer need the address. In the case of a processor/cache bus interface this means that the cache must check the address for a *hit* in its directory before allowing the address cycle to complete.

2.2. Open Collector Signals

Figure 1 illustrates how a broadcast handshake works. All bus signals are open-collector driven and passively terminated. The effect of multiple drivers on a single line is similar to having a number of children stepping on a garden hose. A child's foot on the hose (open-collector driver on) can stop the flow (low logic level), but the removal of one foot will not cause flow to resume (high logic level) as long as someone else is stepping on the hose (another driver is on). Similarly with open-collector signals, if you need to know when the first module reaches a particular state, have it pull the signal low. And if you need to know when "all" modules have reached a particular state, arrange to have them all pulling the signal low initially and wait for the signal to go high. This will only occur when all

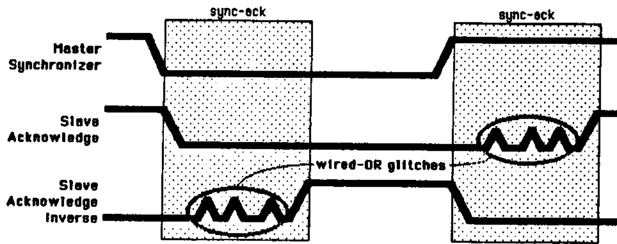

Figure 1: Broadcast handshake on Futurebus.

the drivers have "let go" of the signal. (In summary, *drive low, float high.*)

We do note that an unavoidable perturbation of the signal occurs when one driver releases an open-collector signal that is still being asserted by another driver. The current that is no longer passing through the released driver must now be sunk through a driver that is physically located elsewhere in the backplane. This current switch manifests itself as a small glitch, called a **wired-OR glitch**, whose duration and amplitude are a function of the distance between drivers, and the amount of current that is no longer being supplied by the released driver. The wired-OR glitch problem is deterministically solved by using an asymmetrical inertial delay line, or low pass filter [Gust83]. The exacted penalty on the Futurebus is that broadcast handshaking is 25 nanoseconds slower than single slave transactions. The reward is that broadcast operations are guaranteed to work, no matter how new or old, fast or slow, a particular board may be.

As shown in Figure 2, the current bus master first issues an address, then signals the event by asserting the address strobe, AS*. All other bus modules assert AK* immediately (address acknowledge), but each releases AI* (address acknowledge inverse) and allows it to rise only after it is finished with the address and is ready to go on. Only after AI* has risen may the bus master remove the address from the bus. See [Borr84], [P896] for further details.

2.3. Bus Overview

To summarize some of the more significant features of the electrical protocols of the futurebus, we note the following: (a) Every unit on the bus may examine the address on every address cycle. (b) Only those units participating need monitor data transfer cycles, which can therefore proceed at a high rate. (c) The fact that all units must acknowledge the address cycle implies that any unit has time to signal any sort of "exception", as when it detects a hit in its cache. (d) The nature of the connection level handshake means that more than two parties can participate in a transfer, as when more than one cache updates its copy of a line.

3. The MOESI Model and Class of Protocols

3.1. The MOESI States

From results in [Smit79, 82, 85b] and the discussion in [Good83], it is clear that the best performance and greatest reduction in bus traffic can be obtained with the use of a **copy-back cache.** With such a cache, data modifications are first made to the line in the cache, and then the modifications are written to memory only when the line is replaced. For such a cache, it is possible to observe that each line in the cache may be assigned to one of *five* states, which partition pairwise and can be specified with three bits: *validity, exclusiveness,* and *ownership*; see Figure 3.

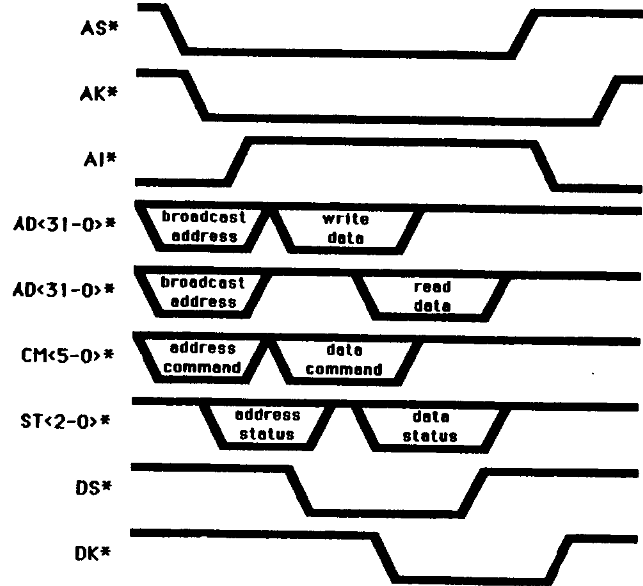

Figure 2: Futurebus parallel protocol.

3.1.1. Validity

Data in shared memory is either **valid** or **invalid**. Shared memory modules will not need to distinguish valid data from invalid data; instead, caches associated with each master will keep track of the invalidity of the data that resides in shared memory. In the absence of information to the contrary, data in shared memory is defined to be valid (e.g. at power-on), although here the term "valid" relates only to the consistency protocol and not to the semantics of the system. The **shared memory image** or **shared image** is

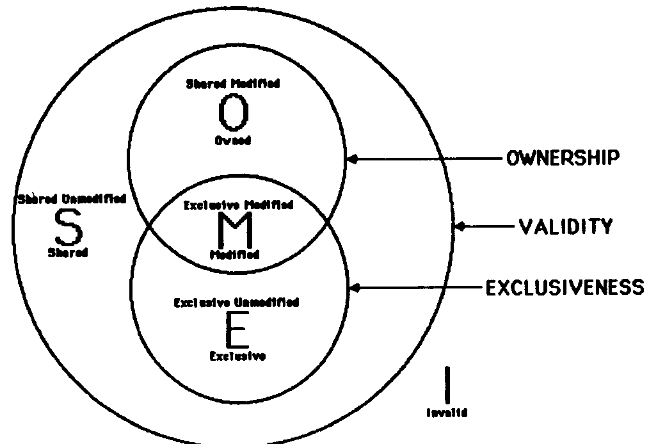

Figure 3: Three characteristics of cached data.

the union of all valid data corresponding to every location of the system address space.

3.1.2. Exclusiveness

The set of all valid data residing in caches can be classified into **exclusive** data and **shared** data. Exclusive data is cached data that is contained in one and only one cache. Non-exclusive data is cached data that may be in more than one cache. The term **shared** is used to describe non-exclusive data even though the number of other cached copies is not known and may be zero. Exclusive data must match the copy in main memory.

3.1.3. Ownership

Valid data residing in a cache may also be characterized as being or not being **owned**. To own data means to assume responsibility for its accuracy or validity for the entire system. Data in main memory may or may not be valid, but that the main memory is not responsible for that determination. That determination is made by the cache that owns that line of data; in particular, if main memory does not contain valid data, the owning cache is responsible for ensuring that: (a) main memory is correctly updated, (b) ownership is passed to another cache, or (c) the owning cache substitutes for main memory in transfers.

All data is said to be owned uniquely either by one and only one cache or by main memory. Therefore, another way to define the *shared memory image* is as the set of *all owned data*; main memory is the default owner.

3.1.4. Resulting State Model

The shared memory is not responsible for tracking the state of the data it holds. All such responsibility resides with the caches. We therefore refer to data stored in caches when referring to the **state** of the data.

There are eight possible ways to apply the three characteristics of cached data to a particular cached data line. However, it is pointless to consider the exclusiveness or ownership of a data line that is known to be invalid. The five remaining states are: (1) Exclusive owned; (2) Shareable owned; (3) Exclusive unowned; (4) Shareable unowned; (5) Invalid. The purpose of the owned category is to distinguish a copy that has been modified from that in main memory. Substituting the term **modified** for owned results in the more familiar terminology: (1) Exclusive modified; (2) Shareable modified; (3) Exclusive unmodified; (4) Shareable unmodified; (5) Invalid. These state labels can be abbreviated by using the salient feature of each state: (1) **M**odified; (2) **O**wned; (3) **E**xclusive; (4) **S**hareable; (5) **I**nvalid.

It is useful to agree beforehand on a consistent terminology. To that end the three above lists are to be considered completely equivalent, with *the last set the preferred terminology*. In addition, *the single-letter abbreviations* **M, O, E, S,** and **I**, *are also appropriate, hence the acronym "MOESI" cache states*. Figure 3 illustrates the way that validity, exclusiveness, and ownership combine to delineate each of the five states.

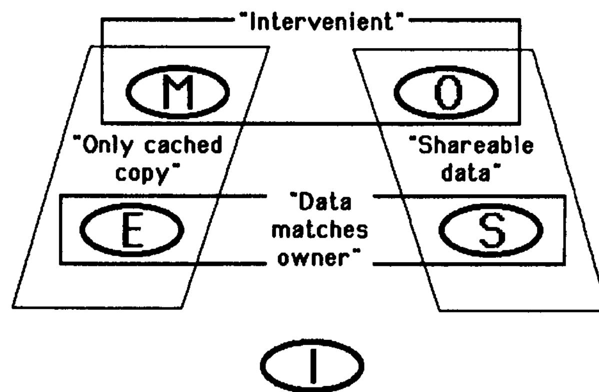

Figure 4: MOESI state pairs.

We can learn something about the utility of cache states in maintaining cache coherence by examining the common qualities of certain **state pairs**. See Figure 4.

M data is also known as *modified, exclusive modified,* and *exclusive owned data.* **O** data is *owned, shareable modified, or shareable owned data.* The common characteristic of M and O data is that the cache holding such data is responsible for the accuracy of that data for the entire system. This means that if the M or O data is not correctly stored in the shared memory, the owner cache must somehow make sure that no other module reads that incorrect data. The term describing this responsibility is **intervention.** M and O data are **intervenient** states.

E data is *exclusive, exclusive unmodified, or exclusive unowned data.* M and E data have in common that they are the only cached copy corresponding to a particular address range of the shared memory image. Suppose that a cache client wishes to change M or E data. Since it knows that no other cache holds a copy, it needs not warn any other caches that the data it holds is about to become invalid.

S data is also known as *shareable, shareable unmodified, or shareable unowned data.* S and E data are both unowned. This means that a cache holding S or E data is not responsible for the integrity of accesses to its data line by other system modules. Note that the S state does not imply that main memory is valid.

S and O data have the common characteristic that they are non-exclusive copies of the data. This means that any attempt by the cache client to locally modify S or O data requires that a message be broadcast to other caches notifying them of the change.

3.2. Signal Lines for Implementing MOESI Consistency

To implement our consistency protocol(s), we need six signal lines on the Futurebus backplane; three are used by the master for the transaction to indicate his intentions; the other three are used for other units on the bus to assert either status or control. To implement versions of other protocols (e.g. Write-Once, Illi-

nois), we also need a seventh signal, BS, (busy), to abort a transaction.

3.2.1. Cache Master Signals

We define the **CA** signal, when asserted, to mean "I am a copy-back cache and at the end of this transaction, I will retain a copy of the referenced data, or I am a write-through cache and have just read this data"; this is called the **cache master** signal. This signal is important so that other units can distinguish operations in which a cache loads data or a copy-back cache retains data, from operations with non-caching units and/or writes by write-through caches.

We define the **IM** signal, called **intent to modify** to mean "in this transaction I will modify the referenced data." This is necessary so that other units can either discard or update their copies of the data.

Finally, we define the **BC** (**broadcast**) signal to mean "if I do modify the data, I will place the modifications on the bus so that you and/or the memory can update itself." If IM is asserted and BC is not, then units holding the data must discard it; the data cannot be updated. Broadcast transactions are generally slower than non-broadcast, due to wired-or glitches (see section 2.2).

3.2.2. Response Signals

The **Cache Hit** (**CH**) line is generally used by units with caches to respond during the address cycle to say: "I have a copy of the referenced data, which I will retain at the end of this transaction." (In a few cases, CH is not asserted by one specific cache in the transaction, so that it can listen and see if CH is asserted by any other cache.) This response is necessary so that the cache performing the transaction and/or owning caches can determine which element of the paired (S,E) and (O,M) states they should enter. When the value of CH doesn't affect anything, i.e. when no other cache would be listening, it is shown as a "don't care" (CH?) in Table 2.

In the event that another cache in the system (other than that of the bus master) is the owner of a data line, it is necessary to use the **DI**, or **Data Intervention** signal, which indicates that the unit asserting this line is the owner of the data, and will preempt a response from memory. I.e. on a read, it will supply the data, and on a write, it will capture the information and update itself.

The **SL** line (**select**) is used on a broadcast transfer by a slave cache that wishes to connect on a transfer and update its own copy of a line. Memory also will assert this line when it participates in a transaction.

Finally, the **BS** (**busy**) line is needed when certain protocols (e.g. Write-Once, Illinois) are to be implemented on Futurebus. This is because *Futurebus currently has no mechanism by which a transfer from one cache to another can also update main memory**. BS is used to abort a transaction and update memory before that transaction can resume.

*An extension to the Futurebus protocol to permit main memory to be updated on a transfer is under discussion.

MOESI Protocol: Result State and Bus Signals

	Local			
Event: note: From State	Read 1	Write 2	Pass 3	Flush 4
M	M	M	E,CA,BC?,W	I,BC?,W
O	O	CH:O/M, CA,IM,BC,W or M,CA,IM	CH:S/E CA,BC?,W	I,BC?,W
E	E	M	—	I
S	S	CH:O/M CA,IM,BC,W or M,CA,IM or S,IM,BC,W* or S,IM,W*	—	I
I	CH:S/E, CA,R or S,CA,R* or I,R**	M,CA,IM,R or Read>Write or I,IM,BC,W*,** or I,IM,W*,** or Read>Write*	—	—

Table 1

3.3. Protocol Class Definition

Given the cache master and response signals defined above, we can define a **class of compatible protocols** supported by Futurebus. The complete class of protocols is defined in Tables 1 and 2. (Where a choice is shown, the first entry is preferred, as we discuss later.) Rather than discuss every entry in those tables, which is neither necessary nor feasible in this limited space, we comment on the more important features of our protocol definition. First we discuss the behavior of a **copy back cache**.

1. A cache with a read miss places the data in S or E states depending on whether anyone else has that information in its local cache (via CH).
2. A unit writing to data in the O or S states must either broadcast the changes (CA, IM, BC) (remaining in or going to O or going to M, depending on CH), or simply invalidate the copies in the other caches (CA, IM, not BC), and go to the M state.

MOESI Protocol: Result State and Bus Signals

	Bus Event					
Event: note: From State	CA,~IM,~BC 5	CA,IM,~BC 6	~CA,~IM,~BC 7	CA,IM,BC 8	~CA,IM,~BC 9	~CA,IM,BC 10
M	O,CH,DI	I,DI	M,DI,CH?	—	M,DI,CH?	M,SL,CH?
O	O,CH,DI	I,DI	CH:O/M, DI	S,SL,CH or I	O,DI,CH?	O,SL,CH
E	S,CH	I	E,CH?	—	I	E,SL,CH? or I
S	S,CH	I	S,CH	S,SL,CH or I	I	S,SL,CH or I
I	I	I	I	I	I	I

Table 2

3. A unit generating a write miss may either request a copy and invalidate other copies simultaneously (CA, IM, Read), or use two transactions to read the line (into state S or E) and then modify it (entering O or M).
4. A cache in an intervenient state (O or M), when it sees a read miss (not IM, not BC, columns 5, 7), must supply the data. When it sees a write by a non-caching master (not CA,IM, not BC, col. 9), it must capture the write. When it sees a broadcast write, by a caching master, (CA, IM, BC, col. 8), it must relinquish ownership and either update itself or invalidate itself; on a broadcast write from a non-copyback unit (col. 10), it must update itself. When it sees a write miss (CA, IM, not BC, col. 6), it must supply the data, and then invalidate itself.
5. A cache in a non-intervenient state (S or E) goes to state S on a read (cols. 5, 7) and raises CH, except when the read is by a non-caching master (col. 7), in which case if in state E, it remains there. On a non-broadcast write (cols. 6, 9), it must invalidate its copy. For a broadcast write (cols. 8, 10), it may either update itself or invalidate its own copy.

Our protocol also works for **write through caches**. A write through cache has two states: V (valid) and I (invalid); a write through cache is not capable of ownership. We equate the V state of the write through cache with the S state of the copy-back cache, and show the protocol definition in Table 1 as well (designated by "*"); we comment on some of the significant features here.

6. A write through cache on a write simply writes through, with or without broadcast. If the cache is to do write allocate, it does a read first, then a write.
7. A write through cache on a read miss does a normal read, asserting CA.
8. On a read on the bus (cols 5, 7), the cache remains valid. On a broadcast write (cols. 8, 10), it can either become invalid or update itself. On a non-broadcast write (cols. 6, 9), it must become invalid, since it is not capable of intervention or ownership.

Our protocol also applies to processors without caches, as is also shown in Table 1 (marked with "**"). Such a processor writes with or without broadcast (as with a write through cache), and reads without asserting CA. A non-caching unit never responds to bus events.

We note some additional alternatives to those shown in Tables 1 and 2. In particular:

9. Any transition of the form CH:O/M can be replaced by O. State M can change at any time to state O, although with a loss of protocol efficiency.
10. Any transition of the form CH:S/E can be replaced by S. State E can change at any time to state S, although with a loss of protocol efficiency.
11. Any transition to or remaining in E or S on a bus event can be changed to I, not CH.
12. The state E may be replaced by the state M, although with a loss of efficiency, due to the now required write-back.

3.4. Protocol Choice, Compatibility, and Variation

We've defined above and in Tables 1 and 2, a class of protocols, such that for many states and events, there is a choice of action. There are some observations we can make about that situation. First, we note that different boards on the bus can implement different protocols, provided that each comes from this class. Second, we note that that each bus user can change the protocol it is using, either statically, dynamically, or can use protocols selectively. For example, a given cache can make some pages copy back, some write through, and some uncacheable (as with the Fairchild CLIPPER [Cho86]). Further, as noted, caches of different types (write through, copy back, etc.) can coexist on the bus simultaneously. As an extreme case, it would introduce no errors if a board were to select an action at each instant from the available set using a random number generator or a selection algorithm such as round robin.

4. Protocol Compatibility and Other Protocols

As noted in the introduction, a number of other bus based cache consistency protocols have been previously defined. The question arises as to whether any (or all) of these preexisting protocols can: (a) be implemented on the Futurebus, and (b) whether there exists an implementation which falls within the class of protocols defined here.

In this section, we look at a number of the earlier protocols (Berkeley, Illinois, Dragon, Firefly, Write-Once) and address those questions. In each case, we do not discuss the protocol definition in any detail, but instead refer the reader to either the original paper or to a recent comparison [Arch85]. Our definitions of the protocols and their mapping into our scheme are presented only to the extent necessary to convey the important points. In particular, we show the definition of each algorithm *only* to the extent necessary to define the algorithm relative to the Futurebus facilities and to its interaction with other caches using the same protocol. If a given board were to use one of these algorithms on a Futurebus system in which other boards were using other protocols, it would be necessary to define the behavior of the board with respect to bus events not generated by its own algorithm. We do not do that here.

4.1. Berkeley Protocol

The "Berkeley Protocol" is so called because it was defined by a group at UC Berkeley [Katz85] as the consistency scheme to be used for the SPUR multiprocessor RISC computer being constructed there. The states in that protocol map into M, O, S and I; there is no state that corresponds to E. The facilities of Futurebus are sufficient to implement the Berkeley Protocol, and a state diagram for the implementation appears in Table 3. The only difference between Table 3 and the protocol as defined in [Katz85] is that in Table 3 the CH signal is generated for compatibility

with our mechanism; the CH signal is not used in [Katz85].

Berkeley Protocol: Result State and Bus Signals

Event: note: From State	Local: Read 1	Local: Write 2	External: CA,~IM,~BC 5	External: CA,IM,~BC 6
M	M	M	O,CH,DI	I,DI
O	O	M,CA,IM	O,CH,DI	I,DI
S	S	M,CA,IM	S,CH	I
I	S,CA,R	M,CA,IM,R	I	I

Table 3

4.2. Dragon Protocol

The "Dragon Protocol" is that used in the Dragon Processor at Xerox PARC, and is defined in [McCr84] and [Arch85]; we rely heavily on the latter due to the vagueness of the definition in the first paper.

The Dragon protocol is implementable almost exactly using the Futurebus features. The one exception is that when a broadcast write is done on the Futurebus, it affects all caches holding the line and also main memory. Such a write for the Dragon protocol does not update main memory; main memory is updated only on a replacement. Extra memory updates, however, cause no incompatibility. An implementation of the Dragon protocol using the Futurebus features appears in Table 4.

Dragon Protocol: Result State and Bus Signals

Event: note: From State	Local: Read 1	Local: Write 2	External: CA,~IM,~BC 5	External: CA,IM,BC 8
M	M	M	O,DI,CH	—
O	O	CH:O/M CA,IM,BC,W	O,DI,CH	S,SL,CH
E	E	M	S,CH	—
S	S	CH:O/M, CA,IM,BC,W	S,CH	S,SL,CH
I	CH:S/E, CA,R	Read>Write	I	I

Table 4

4.3. Write-Once Protocol

The "write once protocol" was defined in [Good83] and was the first bus based consistency protocol invented. The name comes from the fact that the first write to a datum is broadcast in order to invalidate the other copies.

The write-once protocol requires that on an intervenient action, memory be updated at the same time that the intervenient cache supplies the data to the active cache. This is not possible with Futurebus, so an exact implementation is not possible. We replace intervention with an abort (BS), followed by an immediate write back ("push") to main memory; when

Notes on Tables

Format: result state (M, O, E, S, I), bus signals (CA, IM, BC, BS, SL, DI, CH), action (R, W)

*Write Through Cache, **No Cache

Read>Write = two transactions; Read, followed by Write

CA = cache master signal

IM = "intent to modify" - used on address cycle to signal a write (data modify)

BC = "broadcast" - signals intent to broadcast data write

CH = issued by a slave or 3'rd party cache on the bus which will retain the referenced item, CH? = don't care. CH: O/M = If CH then O else M. CH: S/E = If CH then S else E.

DI = response by slave signalling intervention

SL = response by slave or 3'rd party signalling connect on write

BS = "busy" - aborts transaction

W = issue write on bus

R = issue read on bus

Any transition of the form CH:O/M can be replaced by O

Any transition of the form CH:S/E can be replaced by S

Any transition to E or S (on bus events) (and CH) can be changed to I,~CH

1: read by local processor

2: write by local processor

3: push of dirty line by local processor and keep copy

4: push dirty line and discard copy

5: read by cache master on bus (including WT cache)

6: read for modify by cache master (i.e. write miss by copy back cache) or address only invalidate signal

7: read by processor without cache

8: broadcast write by cache master

9: write by processor without cache or write past WT cache

10: broadcast write by non-cache processor or write past WT cache

-- not a legal case. error condition

Write Once Protocol: Result State and Bus Signals

Event: note: From State	Local: Read 1	Local: Write 2	External: CA,~IM,~BC 5	External: CA,IM,~BC 6
M	M	M	BS;S,CA,W	I,DI or BS;S,CA,W
E	E	M	S,CH	I
S	S	E,CA,IM,W	S,CH	I
I	S,CA,R	M,CA,IM,R or Read>Write	I	I

Table 5

the transaction is restarted, memory is up to date and intervention is no longer required.

We show our implementation of the write-once protocol in Table 5. We note that the definition of the algorithm as given in [Good83] and also in [Arch85] is somewhat ambiguous; the varying possible interpretations are reflected by two states in which two actions are connected by "or".

4.4. Illinois Protocol

The Illinois protocol was defined in [Papa84]. There are two of ways in which Futurebus cannot exactly implement this protocol. First, it requires that main memory be updated at the time that a dirty block is passed from one cache to another; we do that function by aborting the transaction (BS), updating memory, and allowing the transaction to restart. Second, all caches are to respond on a match to their local directory, and a bus priority mechanism determines which gets bus access; we cannot permit that in our protocol. Our implementation has either an intervenient cache or main memory respond (which is always a unique respondent); caches in states S and E never respond.

It is possible to map the states of the Illinois protocol into our states, but we note that the S state has a different meaning. The Illinois protocol defines the S state as consistent with memory; that is not the case for the protocol as we have defined it.

Our implementation of the Illinois protocol, as supported by the Futurebus, is shown in Table 6.

4.5. Firefly Protocol

The "Firefly Protocol" refers to a consistency scheme being implemented at DEC SRC; the only definition available is provided in [Arch85].

The Firefly protocol requires that when an intervenient cache provides data, memory be updated. We again do that function by aborting the transaction, updating main memory, and restarting. This also means that their S and E states are consistent with main memory, whereas our S state is consistent with the owner but not necessarily with main memory. Finally, all of their caches respond on a read (as with the Illinois protocol), which we do not permit; we leave it to main memory or the intervenient cache to respond, as appropriate.

Our implementation of the Firefly protocol, as supported by the Futurebus, is shown in Table 7.

5. Other Considerations for a Futurebus Consistency Standard

There are some other issues in defining a cache consistency standard for a standard bus, other than the protocols to be used. We note those here very briefly, without exploring them in detail.

5.1. A Standard Line Size

The line size (block size) is the size of the data unit in the cache. Lines are transferred in their entirety to a cache, where the line contents are associated with an address tag. The entire discussion above has implicitly assumed that the line size is constant across all caches in the system.

If the line size is not constant throughout the system, some very difficult problems can arise. For example, let cache A (with a line of 64 bytes) do a read. Cache B (with a line of 32 bytes) has *part* of that line

Illinois Protocol: Result State and Bus Signals

	Local		External	
Event: note: From State	Read 1	Write 2	CA,~IM,~BC 5	CA,IM,~BC 6
M	M	M	BS;S,CA,W	BS;S,CA,W
E	E	M	S,CH	I
S	S	M,CA,IM	S,CH	I
I	CH:S/E, CA,R	M,CA,IM,R	I	I

Table 6

resident in state M. Cache B is therefore required to supply part of the line requested by cache A, but where is the rest of the line to come from?

The opinion of the P896.2 working group is that the difficulties of managing a non-constant line size are such as to require that a given system standardize on a given line size, and that further, it is the responsibility of the working group to recommend a size. A recommended line size has not yet been selected, but we refer the reader to [Smit85c] for a discussion of the data and methodology to be used for such a recommendation.

Firefly Protocol: Result State and Bus Signals

	Local		External	
Event: note: From State	Read 1	Write 2	CA,~IM,~BC 5	CA,IM,BC 8
M	M	M	BS;E,CA,W	—
E	E	M	S,CH	—
S	S	CH:S/E, CA,IM,BC,W	S,CH	S,SL,CH
I	CH:S/E, CA,R	Read>Write	I	I

Table 7

There is also the problem of supporting **sector caches** [Hill84]. The implications of that design have not been fully explored at this time, and it is undetermined whether the address sector size, the transfer subsector size or both must be standardized. (The latter almost certainly needs to be fixed, for the same reasons noted above. Consistency status also appears to be necessarily associated with the transfer subsector, rather than the address sector.)

It is also worth noting the problem of "line crossers"; i.e. a processor operation which makes a reference which overlaps 2 or more lines. It should be clear that the processor/cache interface must be able to treat this as a separate transaction for each line

involved, and to generate bus transactions on that basis.

5.2. Performance Issues and Enhancements

As noted earlier in this paper, the preferred protocol choice (from Tables 1,2) was always the first entry in a given box. That preference is based on results from [Arch85], which presents the results of simulations which are based only on a model of program behavior [Dubo82]. That work needs to be supplemented with experiments based on real multiprocessor shared memory address traces.

One of the more interesting observations from [Arch85] was that it was desirable to broadcast writes to other caches rather than to invalidate them, if those other caches have the line in them. A refinement on that is to have a cache examine the replacement status of a line written by another cache. If the line is quite recently used (e.g. most recently used element of two element set), it can be updated, and if it is nearing time for replacement (e.g. least recently used element of two element set), it can be discarded. (See [Puza83] for a related idea.)

We also note that the preferred protocol is sensitive to the implementation of the bus, the memory and the caches. Changes in their relative performance can change the cost of various bus operations (e.g. memory read, intervenient cache read, etc.) and change the preferred actions.

6. Conclusions and Summary

In this paper, we have discussed the problem of defining a cache consistency protocol for a standard bus, in particular the IEEE Futurebus. We have defined a class of compatible protocols, such that each cache in the system may implement one of the protocols in this class and still maintain consistency with other caches implementing different (compatible) protocols. This permits the coexistence of copy back caches, write through caches and non-caching boards in the same system.

We have shown that a number of previously published protocols (Berkeley, Illinois, Dragon, Firefly, Write Once) can be supported (either as defined or with minor modifications) on the Futurebus, and we have defined them in that context.

Finally, we have very briefly discussed some other issues relating to a standard caching mechanism, including that of a standard line size and the support of sector caches.

There are a number of aspects of this work that must be continued. All implications of caching standardization must be fully explored, including line size, sector caches, and how one might implement a system with *multiple* buses and still maintain consistency. Further research is required (with better data than [Arch85]) to determine the best performance choice (or cost/performance choice) to be made in our class of compatible protocols. Proper mechanisms must also be defined for issuing commands across the bus to cause other caches to become consistent with main memory.

Acknowledgements

The work described in this paper comes out of discussions of the P896 (Futurebus) committee and the P896.2 (caching) working group. We want to thank the members of those groups, particularly Paul Borrill, Jim Goodman, Mark Papamarcos and Dave Gustavson for their contributions. Thanks also to Mark Hill and Susan Eggers who read and commented on this paper.

Bibliography

[ANSI86] ANSI/IEEE Standard 960 - 1986 Fastbus Modular High Speed Data Acquisition and Control System

[Arch84] James Archibald and Jean-Loup Baer, "An Economical Solution to the Cache Coherency Problem", Proc. 11'th Ann. Symp. on Comp. Arch., June, 1984, Ann Arbor, Michigan, pp. 355-362.

[Arch85] James Archibald and Jean-Loup Baer, "An Evaluation of Cache Coherence Solutions in Shared-Bus Multiprocessors", Tech. Rpt. 85-10-05, October, 1985, Computer Science Dept., University of Washington, Seattle, Wash.

[Bala84] R. V. Balakrishnan, "The Proposed IEEE 896 Futurebus - A Solution to the Bus Driving Problem", IEEE Micro, August, 1984, pp. 23-27.

[Borr84] Paul Borrill and John Theus, "An Advanced Communication Protocol for the Proposed IEEE 896 Futurebus", IEEE Micro, August, 1984, pp. 42-56.

[Borr85] Paul L. Borrill, "Microstandards Special Feature: A Comparison of 32-Bit Buses", IEEE MICRO, December, 1985, pp. 71-79.

[Cens78] Lucien Censier and Paul Feautrier, "A New Solution to Coherence Problems in Multicache Systems", IEEETC, C-27, 12, December, 1978, pp. 1112-1118.

[Cho86] James Cho, Alan Jay Smith and Howard Sachs, "The Memory Architecture and Cache and Memory Management Unit for the Fairchild CLIPPER Processor", February, 1986, submitted for publication. (Available from Fairchild Advanced Processor Division, 4001 Miranda Ave., Palo Alto, Ca., or from Alan Smith, UC Berkeley.)

[Coop83] Steve Cooper, "Multibus Continues to Evolve to Meet the Challenges of the VLSI Revolution", Proc. NCC 1983, pp. 497-501.

[Dubo82] M. Dubois and F. Briggs, "Effects of Cache Coherency in Multiprocessors", IEEETC C-31, 11, November, 1982, pp. 1083-1099

[Elay85] Khaled A. El-Ayat and Rakesh Agarwal, "The Intel 80386 - Architecture and Implementation"', IEEE MICRO, December, 1985, pp. 4-22.

[Emer41] Ralph Waldo Emerson, Essays: First Series, 1841, "Self Reliance".

[Fair85] Fairchild, "CLIPPER Module Product Description", Fairchild Camera and Instrument Corporation, October, 1985.

[Fish84] Wayne Fischer, "The VMEbus Project", Proc. IEEE Compcon, February, 1984, pp. 376-378.

[Fran84] Steven J. Frank, "Tightly Coupled Multiprocessor System Speeds Memory Access Times", Electronics, January 12, 1984, pp. 164-169.

[Good83] James R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic", Proc. 10'th Ann. Int. Symp. on Comp. Arch., June, 1983, Stockholm, Sweden, pp. 124-131.

[Gust84] David B. Gustavson, "Computer Buses- A Tutorial", IEEE MICRO, August, 1984, pp. 7-22.

[Gust83] David B. Gustavson and John Theus, "Wire Or Logic on Transmission Lines", IEEE MICRO, 3, 3, June, 1983, pp. 51-55.

[Hill84] Mark Hill and Alan Jay Smith, "Experimental Evaluation of On-Chip Microprocessor Cache Memories", Proc. 11'th Ann. Symp. on Computer Architecture, June, 1984, Ann Arbor, Michigan, pp. 158-166.

[Katz85] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, "Implementing a Cache Consistency Protocol", Proc. 12'th Ann. Int. Symp. on Comp. Arch., June, 1985, Boston, Mass, pp. 276-283.

[Mate85] Richard Mateosian, "National's NS32332 CPU: A Graceful Extension of the Series 32000", Profession Program Session Record, Wescon/85, November, 1985, San Francisco, Ca., Session 1: 32-Bit Microprocessors - Part I.

[McCr84] Edward M. McCreight, "The Dragon Computer System: An Early Overview", June, 1984, Technical Report, Xerox PARC.

[Moto84] Motorola Corporation, "MC68020 Technical Summary", 1984.

[Mous86] J. Moussouris, L. Crudele, D. Freitas, C. Hansen, E. Hudson, R. March, S. Przybylski, T. Riordan, C. Rowen, and D. Van't Hof, "A CMOS RISC Processor with Integrated System Functions", Proc. IEEE Compcon, March, 1986, pp. 126-132.

[P896] IEEE P896 Draft Standard, Backplane Bus (Futurebus)

[P1014] IEEEP1014 Versatile Backplane Bus (VME Bus)

[Papa84] Mark Papamarcos and Janak Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories", Proc. 11'th Ann. Int. Symp. on Comp. Arch., June, 1984, Ann Arbor, Michigan, pp. 348-354.

[Phil85] David Phillips, "The Z80000 Microprocessor", IEEE MICRO, December, 1985, pp. 23-36.

[Puza83] T. R. Puzak, R. N. Rechtschaffen and K. So, "Managing Targets of Multiprocessor Cross Interrogates", IBM Tech. Disc. Bull., 25, 12, May, 1983, p. 6462.

[Rudo84] Larry Rudolph and Zary Segall, "Dynamic Decentralized Cache Schemes for MIMD Parallel Architectures", Proc. 11'th Ann. Int. Symp. on Comp. Arch., June, 1984, Ann Arbor, Michigan, pp. 340-347.

[Smit79] Alan Jay Smith, "Characterizing the Storage Process and its Effect on the Update of Main Memory by Write-Through", JACM, 26, 1, January, 1979, pp. 6-27.

[Smit82] Alan Jay Smith, "Cache Memories", Computing Surveys, 14, 3, September, 1982, pp. 473-530.

[Smit84a] Alan Jay Smith, "CPU Cache Memories", to appear in *Handbook for Computer Designers*, ed. Flynn and Rossman.

[Smit84b] Alan Jay Smith, "Trends and Prospects in Computer System Design", part of proceedings of a Seminar on High Technology, at the Korea Institute for Industrial Economics and Technology, Seoul, Korea, June 21-22, 1984. Available as UC Berkeley CS Report UCB/CSD84/219. Verbatim transcript of speech published in "Challenges to High Technology Industries", Korea Institute for Economics and Technology, pp. 79-152.

[Smit85a] Alan Jay Smith, "Problems, Directions and Issues in Memory Hierarchies", Proc. 18'th Annual Hawaii International Conference on System Sciences, January 2-4, 1985, Honolulu, Hawaii, pp. 468-476. Also available as UC Berkeley CS Report UCB/CSD84/220.

[Smit85b] Alan Jay Smith, "Cache Evaluation and the Impact of Workload Choice", Report UCB/CSD85/229, March, 1985, Proc. 12'th International Symposium on Computer Architecture, June 17-19, 1985, Boston, Mass, pp. 64-75.

[Smit85c] Alan Jay Smith, "Line (Block) Size Selection in CPU Cache Memories", June, 1985. To appear, IEEETC. Available as UC Berkeley CS Report UCB/CSD85/239.

[Smit85d] "CPU Cache Consistency with Software Support and Using "One Time Identifiers"", Proc. Pacific Computer Communication Symposium, Seoul, Republic of Korea, October 22-24, 1985, pp. 142-150.

[Smit86] "Bibliography and Readings on CPU Cache Memories", February, 1986, to appear, Computer Architecture News.

[Tang76] C. K. Tang, "Cache System Design in the Tightly Coupled Multiprocessor System", Proc. NCC, 1976, pp. 749-753.

[Texa83] Texas Instruments, "NuBus Specification", 1983, TI-2242825-0001.

[Vern85] Mary K. Vernon and Mark A. Holliday, "Performance Analysis of Multiprocessor Cache Consistency Protocols Using Generalized Timed Petri Nets", Tech. Rpt., University of Wisconsin, Computer Science Dept., 1985. To appear, Proc. Sigmetrics '86.