An Experimental System for Evaluating Cache Coherence Protocols in Shared Memory Multiprocessors

Peter John Ashenden

Department of Computer Science
The University of Adelaide

Submitted for the Degree of Doctor of Philosophy
in
January 1997
Revised August 1997

Contents

List of Figures

List of Tables

Abstract

This thesis examines cache coherence protocols designed for use in bus connected shared memory multiprocessors. The cache coherency problem is discussed, and several previously published protocols are described in a new uniform framework, allowing ready comparison between them to be made. The issue of protocol correctness is also addressed. The protocol mechanisms proposed for the IEEEE P896.2 Futurebus are described, and the way they may be used to implement the published protocols is illustrated. Two approaches for verifying correctness of the Futurebus mechanisms are described. A brief survey of techniques for evaluating relative performance of cache coherence protocols is presented, covering three main techniques: analytical, simulation based, and measurement of real systems. It is argued that the last of these is the most accurate, and that the results of such measurements are needed to validate evaluations based on the other two techniques. A brief description is presented of an early prototype multiprocessor, the Leopard-1, and a detailed description is presented of a full-scale multiprocessor system, Leopard-2. The Leopard-2 is an experimental platform which includes programmable cache controllers, designed to allow performance measurements of cache coherence protocols. Attention is focussed on the design of the cache attached to each processor, and the way in which the coherence protocols are implemented is described. A detailed behavioural model of the programmable cache controller is presented. The model is driven by two workloads: a synthetic workload to exercise specific aspects of system behaviour, and a pseudo-random workload to provide comprehensive test coverage. The model is used to verify correct maintenance of coherence by the caches operating under different coherence protocols. The thesis concludes with

a discussion of ways in which the experimental platform is used to measure relative performance of coherence protocols.

Statement

I, the author of this thesis, hereby declare that this thesis does not contain material which has been accepted for the award of any other degree or diploma in any University, and that, to the best of by knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

I consent to this thesis being made available for photocopying and loan if accepted for the award of the degree for which it is submitted.

Peter J. Ashenden

Acknowledgments

The design work on the Leopard multiprocessors reported on in this thesis was done as part of a large project involving numerous people in a variety of capacities. I would like to thank them all for being part of the team, and making the work possible.

Firstly, I would like to thank the engineering staff on the Leopard Project, Chonghe "Bobby" Fang, Rob Gerhofer, Ken Howard and Gordon Slater, for their tireless efforts on a project that turned out larger than we had all imagined. Thanks also to Peter Daly and Werner Dorfl for technical support and "nuts and bolts" help, and to Peter Hawryszkiewycz and Mike Petty for project management support.

The project also had significant industry involvement. Thanks are due to Alex Stanco, David Knight and Ken Howard (then) of Quentron Optics Pty Ltd, who set up the initial collaborative effort that got the Leopard Project under way. During the life of the project other industry supporters included: National Semiconductor (Aust). Pty Ltd for donations of equipment, software and electronic components; National Semiconductor Corporation, for travel support, donations of components, advance information and technical support, and access to numerous personnel beyond the usual support channels; Silvar-Lisco Corporation, for CAE software and technical support; Quest International Computers Pty Ltd, for CAE software and technical support; NSD Pty Ltd and the George Brown Group, for donation of electronic components and for keeping the project up to date on component information.

The support of the several agencies who provided research funding for the Leopard Project is also gratefully acknowledged: CSIRO Division of Information Technology, Defence Science and Technology Organisation, The Australian Government through the Australian Research Council, and the Australian Telecommunications and Electronics Research Board.

Thanks are due to a number of colleagues, who, though not directly involved in the Leopard Project, were always interested in what was going on, and eager to discuss ideas and offer helpful suggestions. They are Francis Vaughan, Kevin Maciunas and David Hodson. Special thanks also to Chris Marlin, who acted as assistant supervisor for my thesis research, and who also took a keen interest in assisting with project management and linking this project with other research projects under way in the department.

Finally, and most importantly, I thank my supervisor, Chris Barter, not only for seeing me through this thesis and the research behind it, but for his commitment to the belief that hardware-based research is vital, and for creating an environment to support it.

Dedication

I dedicate this work to my wife, Katrina, without whose love, support and encouragement, it would not have come to fruition.

Chapter 1
Introduction

1.1 Workstations and Networks

A recent trend in the evolution of computer systems has been the increased use of workstation computers in a networked environment. This trend has been brought about by the increasing performance per unit cost of the components comprising a workstation or a network server. For example, VLSI processor components have improved in performance by a factor of ten in the last three years, with the cost per part staying approximately constant. Semiconductor memory components have increased in capacity by a factor of four every three years with approximately constant cost per component, and disk storage components have followed a similar trend.

These developments have made it feasible and desirable to distribute computing resources amongst individual users, in the form of workstations. A typical environment now consists of a number of workstations located in applications areas (e.g. offices or laboratories). Each workstation has its own processing and memory resources, and uses a medium- to high-resolution graphics display to present a highly interactive user interface. The workstations are connected to a local network, and use this to communicate with each other, and with resource servers such as file servers, network routers and application-specific computation servers (e.g. simulation accelerators).

There are a number of application areas which benefit from the advantages of a workstation environment, but which require more processing performance than current workstations can provide. Some examples of such applications are Computer Aided Engineering (CAE), Computer Aided Software Engineering (CASE), and document preparation and management systems. CAE of electrical and mechanical systems require large amounts of computing resource for behavioural and physical simulation, and for physical design (e.g. in circuit routing or design rule checking). Software Engineering environments require resources for propagating incremental changes to large software systems, and for analyzing and compiling code. Document processing requires resources for page layout and formatting, indexing and searching.

In a network environment comprising workstations and servers, the additional processing performance required could be added either in individual workstations, or in the servers; the choice made depends on the particular application. For example, processing associated with display transformations would probably be best done in the workstation, whereas searching documents would best be performed in the server which stores the document files. In both cases, processing performance can be increased in one of two ways: either the performance of the single processor can be increased, or multiple processors can be used to form a multiprocessor computer. The former choice has the disadvantage that the cost of a monoprocessor is not linearly related to its performance; that is, it costs more than twice as much to make a monoprocessor twice as fast. For this and other reasons, multiprocessors are the preferred choice.

1.2 Multiprocessor Architectures

Multiprocessors are often divided into two classes: Single Instruction stream/Multiple Data stream (SIMD), and Multiple Instruction stream/Multiple Data stream (MIMD) [20]. In a SIMD multiprocessor, the processors all execute the same instructions in lock-step, but operate on different elements of data. SIMD computers can be used to great advantage in particular applications, such as signal processing or large numerical problems, but they are not suitable for general purpose processing. MIMD computers, on the other hand, are more general purpose in nature, as they do not require great regularity in data or algorithm to take advantage of potential parallelism in computation.

MIMD multiprocessors can be further classified according to two criteria:

- whether memory is shared between processors or private to each processor, and
- the interconnect network between processors and/or memory.

In a shared memory multiprocessor, all of the processors can access memory using a common address space. The code and data for each task can be stored in shared memory, and shared variables used for inter-task communication. The physical memory may be centralized, or distributed amongst the processors.

In a multiprocessor with private memory, each processor stores the code and data for tasks it runs in its own private memory. Communication with other tasks is done using message passing over the interconnection network. Thus this model is best suited to software written using a message passing paradigm, although data sharing can be implemented at some software cost.

There are a number of alternatives for the interconnection, the most suitable for an application depending on the communication patterns exhibited. Interconnection topologies can be broadly divided into three classes: link connected, switching network, and bus connected.

In a link connected architecture, elements are connected using point to point communications channels. Examples of such architectures include hypercubes, meshes and rings. If an application can be partitioned in such a way that processing tasks can be mapped onto the processors with relatively little communication required, then such an architecture can be used to advantage. It also has the advantage that it scales well to a large number of elements. However, as a general purpose machine, the architecture may impose significant communication and data routing overhead. Furthermore, it is very difficult in general to perform allocation of tasks to processors, and the alternative, tasks migration, is expensive in terms of performance.

Switching network architectures use routing switches to transfer data between elements. Possible topologies for the interconnect include crossbars, shuffle exchange networks, trees and hypertrees. These architectures have the advantage that they scale well, however the cost of the switching network is significant. For this reason, switching network architectures are currently only seen in supercomputers.

The third interconnection scheme, the bus connected architecture, consists of elements connected by a broadcast bus. Each element may be a processor, a memory, an I/O interface, or a combination these. It is a highly cost effective and general architecture, and is particularly well suited to workstations, however, it has the disadvantage of not scaling to interconnect a large number of processors.

1.3 Bus Connected Shared Memory Multiprocessors

This thesis concentrates on the bus connected shared memory architecture, illustrated in Figure 1-1. It consists of a number of processors connected to the shared memory with a broadcast bus. The illustration shows the shared memory as centralized, but in general, it may be distributed amongst the processors. The illustration also shows how I/O devices can be attached.

This system organization has several advantages in the context of a workstation environment. Firstly, it is a very general structure. Any inter-task communication mechanism, such as remote procedure call, message passing, or shared data access, can be readily and efficiently implemented using the shared memory system. Furthermore, tasks can be allocated to run on any processor, with little cost in task migration.

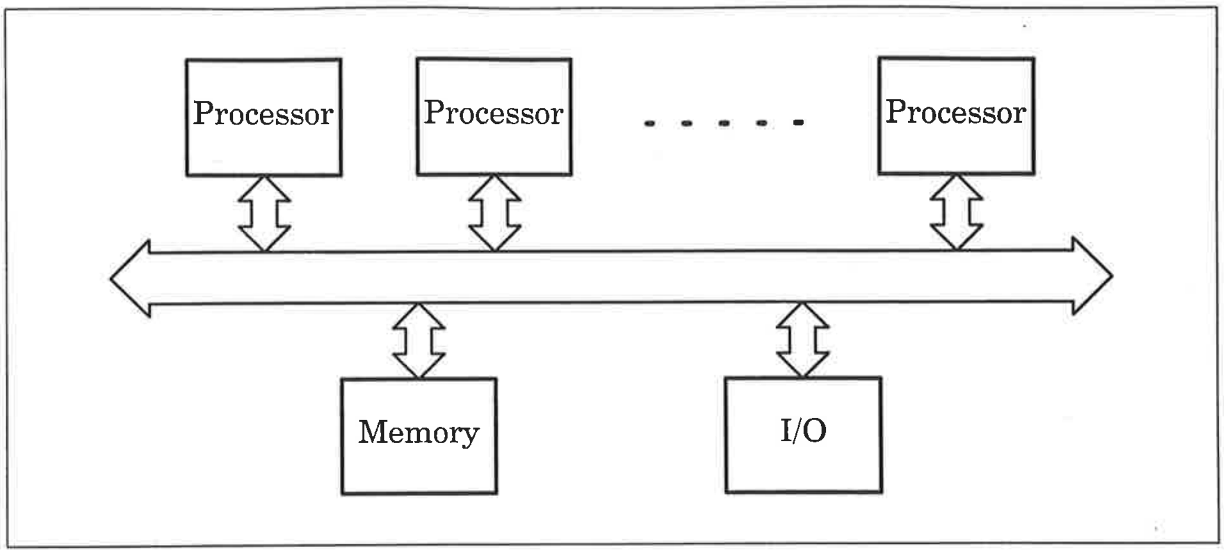


Figure 1-1. The bus connected shared memory architecture.

This greatly simplifies scheduler design, and allows for easier subdivision of a program into tasks and mapping of tasks onto the physical machine. In particular dynamically created tasks can be easily handled.

The second significant advantage of this organization is that it is incrementally expandable, at the granularity of individual processors, in a way which is transparent to applications tasks. If a system is expanded, an application need not be reconfigured to take advantage of the added resources; the advantage may be gained automatically. A related point is that if a processor fails, it can be removed from the configuration, and the remainder of the system can continue to operate, albeit at a degraded level of performance. Thus a high-availability system can readily be constructed.

The third advantage, and one that is especially important in the context of a workstation environment, is that the cost of interconnection of components is low. The interconnection between processors and memory is a passive backplane bus, requiring only the wiring on the backplane and an interface device on each connected module. This is in contrast with other interconnection networks, in which active routing switches are required within the interconnection network, thus adding to the cost.

Fourthly, the bus and shared memory structure allows for simplified connection of interfaces to external devices. Such interfaces can be designed to integrate much more closely with the software environment implemented on the machine. For example, they can be memory mapped, accessible to software in the same address space as memory. Alternatively, they can act as additional processors, using the same inter-task communications mechanism as user and operating system tasks.

The primary disadvantage of the architecture is that it does not scale well to a large number of connected elements. This is because the bus is a shared resource with fixed capacity. When the communication bandwidth required exceeds the capacity of the bus, no further performance gain can be achieved. However, current technology allows a system to be constructed which supports of the order of ten to twenty processors on a single bus. Further scaling can be achieved by constructing a network of buses connected by bus relays or by intelligent inter-bus communication interfaces.

An important aspect of the bus connected shared memory architecture is its use of cache memories attached to each of the processors. In addition to serving their conventional purpose in the memory hierarchy (providing fast access to frequently used data), the caches reduce the load placed on the bus and the shared memory by each processor, thus allowing more processors to be used in a system. Chapter 2 describes the cache coherence problem that arises, and surveys some protocols for maintaining coherence.

Since the operation of the caches is so important in the shared memory multiprocessor architecture, it is desirable to choose a coherence protocol that maximizes performance for the intended applications. Performance of a protocol can be assessed using three methods. Firstly, a mathematical model of a cache system can be devised, with parameters representing attributes of the system. Performance questions can be answered by solving for unknown parameter values. Secondly, a simulation model of a system can be constructed, and program address traces used to stimulate the simulation. Statistical analysis of events in the simulation can then be used to answer performance questions. Thirdly, prototype hardware can be constructed to implement one or more coherence protocols, and the properties of the system running some workload can be measured. These three approaches are surveyed in the latter part of Chapter 2 of this thesis.

1.4 Contribution

One of the major contributions of this thesis is a new and comprehensive survey of previously published cache coherence protocols for bus connected shared memory multiprocessors. This is presented in Chapter 2. To clarify the descriptions of the protocols, a uniform descriptive framework is adopted, an earlier version of which was developed by the author as a contribution to the IEEE Futurebus Cache Coherence Task Group in 1987. Other authors have surveyed cache coherence protocols (for example, [2]), but not in a uniform framework allowing direct comparisons to be made, nor highlighting the similarities and differences. This has made it unnecessarily difficult for system designers and other researchers to understand, evaluate and implement cache coherence protocols. On the other hand, using a uniform descriptive framework, the differences between the protocols are clearly delineated, which is important for a comparative evaluation. Furthermore, the uniform description leads directly to identification of a set of primitive bus protocol mechanisms, also identified in Chapter 2, for implementing the cache coherence protocols. These, in turn, lead to the design of a cache that can be programmed to implement the different cache coherence protocols, thus allowing experimental evaluation of the protocols in real hardware.

Another major contribution of this thesis is a demonstration of the use of behavioural modelling techniques for specification of cache coherence protocols. Chapter 2 includes a description of the IEEE P896.2 cache coherence protocol mechanisms, and Appendix C shows a technique for specifying cache coherence protocols using the P896.2

mechanisms with an information structure model in a hardware description language. This kind of specification is more readily understood than mathematical models by system designers who ultimately implement cache coherence protocols in real computer systems, yet it preserves formality of specification through the rigour of specification of the hardware description language.

The focus of the experimental work described in this thesis is a prototype shared memory multiprocessor system, the Leopard-2, designed and constructed as part of the Leopard Project at the University of Adelaide. The Leopard-2 was designed to allow a number of cache coherence protocols to be implemented and their performance measured in a controlled environment on real hardware under real workloads. The author's role in this project involved developing the initial concept and the Leopard architectural framework, designing the Leopard-2 at the system level, determining the internal organization of each of the main components, participating in the detailed engineering design, construction and testing, and taking on a major part of the project management. Chapter 3 of this thesis firstly describes the Leopard-1, a predecessor to the Leopard-2, used to gain experience with multiprocessor design and to test a number of concepts in bus design to support cache coherence protocols. Chapter 3 then describes the components of the Leopard-2, focussing on the cache attached to each processor.

The Futurebus cache coherence mechanisms, described in Chapter 2, make it feasible to implement a cache controller that is programmable. Such a controller can be reprogrammed to execute any of the cache coherence protocols described in this thesis. Chapter 4 describes the design of a programmable cache controller and illustrates how the cache coherence protocols discussed in Chapter 2 can be implemented by the programmable cache design. It then presents a model of the Leopard-2 system developed using the hardware description language VHDL. The purpose of this model was to verify the hardware design and to specify the detailed behaviour of the programmable cache controller. Simulation was performed using synthetic traces of processor activity

to ensure that the caches operate and interact correctly, and that cache coherence is maintained according to the programmed cache coherence protocol.

The thesis concludes with a discussion of the use of the Leopard-2 as an experimental vehicle for evaluating cache coherence protocols.

Chapter 2
Cache Coherence and Performance Evaluation

2.1 Caches and Cache Coherence

In the past, high performance processors have generally incorporated memory caches. The reason for this has been to avoid the delay involved in fetching data from main memory. Furthermore, the data transfer rate required by the processor has been greater than the main memory bandwidth, and a cache, being constructed with faster memory devices, was able to operate at the required bandwidth.

These reasons for using memory caches still exist in a shared memory multiprocessor architecture, but there is an additional benefit obtained if a cache is attached to each processor. That is that the frequency of accesses from each processor to main memory is substantially reduced. This means that a system with given main memory and bus bandwidth can effectively support a larger number of processors, and so the aggregate system throughput is increased.

To illustrate this, consider a system in which the system bus has a bandwidth of 200 Mbytes/s, each processor requires access to memory at a rate of 200 Mbytes/s, and the cache miss rate is 10%. A first order approximation would indicate that, without caches, each processor would require all of the bus bandwidth, and so the maximum number of processors which could be supported by the bus is one. Any further processors added to the system would not add to aggregate throughput, since there would not be bus capacity to allow the extra accesses to shared memory. However, with caches operating, only 10% of each processor's memory accesses would require access to the shared memory. Hence each processor would require only one tenth of the bus bandwidth, and so up to ten processors may be used effectively.

When data is shared between programs executing on different processors in a shared memory multiprocessor, there is a requirement to ensure that processors have coherent and consistent views of the shared data. Hennessy and Patterson summarize the distinction between coherence and consistency as follows ([26], p. 657):

> Coherence and consistency are complementary: coherence defines the behaviour of reads and writes to the same memory location, while consistency defines the behaviour of reads and writes with respect to accesses to other memory locations.

They define three conditions for coherence. First, if a processor P writes a value to a location, and no other processor writes to the location, P always receives the written value on subsequent reads from the location. In other words, from one processor's point of view, memory operations are performed in program order. Second, if processor P_1 writes a value to a location, and no other processor writes to the location, eventually some other processor P_2 will receive the written value when it reads the location. In other words, writing to memory has an observable effect from other processors' points of view. Third, successive writes to a location by any two processors are seen in the same order by all processors. In other words, writes are serialized, so that the final value seen in a location by all processors is the same.

Hennessy and Patterson describe memory consistency as the issue of when writes from one processor are observed by other processors. In particular, consistency addresses the issue of relative ordering of reads and writes to different locations as seen by different processors in a multiprocessor system. A common requirement is sequential consistency, that is, the results of execution of any individual processor's program appear to all processors in the order specified by that processor's program [45]. It is not necessary to specify the relative order of appearance of results of different processors' programs; where the relative order is important, software synchronization constructs such as semaphores are used. More relaxed orderings include total store order (also called processor consistency), partial store order, weak ordering, and release consistency. These are described in [26], and allow successively less strict maintenance of ordering of reads and writes issued by one processor.

The reason for using relaxed memory consistency models is to allow greater parallelism in the memory system, in order to support greater fine-grain parallelism in the processors. Even when a relaxed consistency model is used, there is still the requirement for coherence, since coherence ensures that the memory behaviour conforms with program order of reads and writes seen by an individual processor. Since this thesis focuses on coherence, the issue of memory consistency will not be discussed further. The interested reader is referred to [26] and the references cited therein for a detailed discussion.

There have been a number of cache coherence strategies for bus-based systems published in the literature. The view of coherence in these strategies is that the value returned to a processor when it performs a read operation from an address is the value written by the most recently performed write operation to that address [15]. This is ensured by equipping each cache with a bus 'snoop', which monitors transactions between other processors and shared memory. It is the responsibility of the snoop to ensure that its cache is maintained in a state which would not violate coherence. To do this, the snoop may need to modify state information in its cache, or to participate in bus transactions as a third party.

To illustrate the requirement for cache coherence, and to show how a system might maintain coherence, consider an example. Suppose a system includes three processors (P_1, P_2 and P_3), each with a copy-back cache (C_1, C_2 and C_3 respectively), and a shared

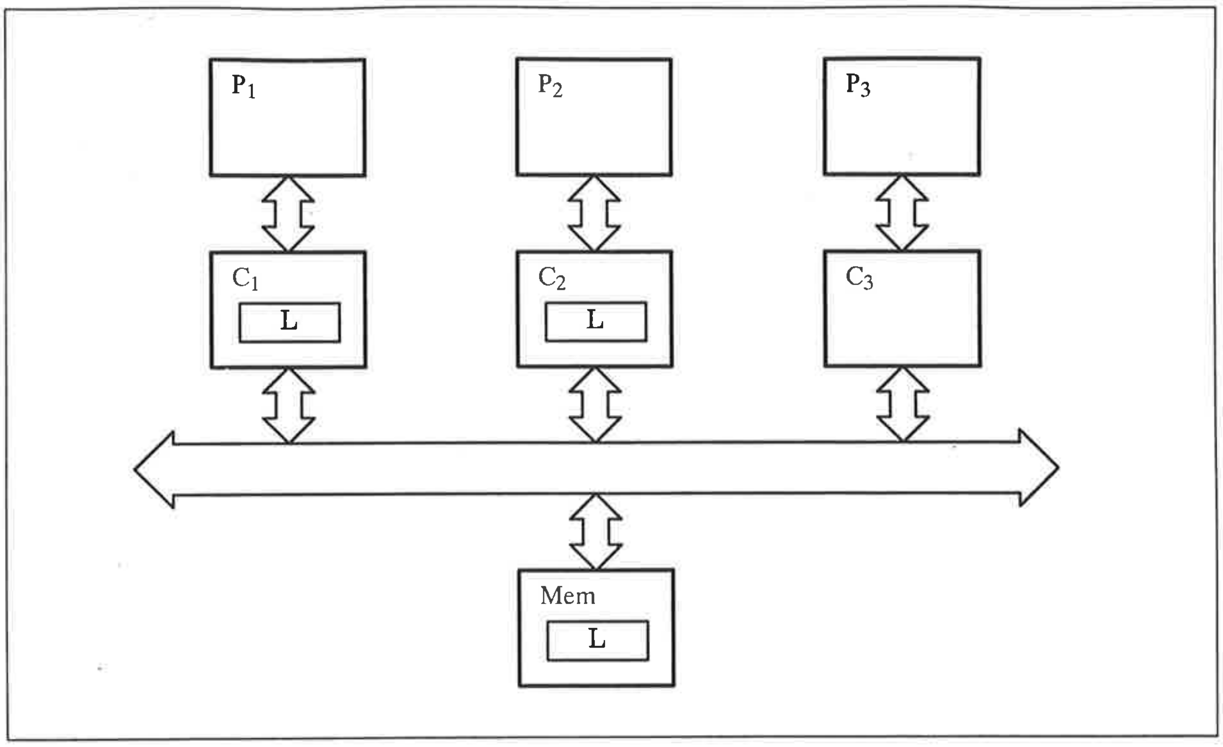


Figure 2-1. Example of cache coherence.

memory containing a line of data L. Suppose firstly that P_1 and P_2 each read L, causing copies to be fetched into C_1 and C_2, as shown in Figure 2-1. If P_1 were to write a new value to L, denoted by L′, the copy in C_1 would be modified, and the copies in C_2 and shared memory would be out of date. A read access by P_2 or P_3 would return the old value, not the most recently written value.

In order to maintain coherence, C_1 may do one of a number of things. One alternative is that C_1 may broadcast the new value L′ to all caches, and possibly to shared memory as well. This is an example of a *write-broadcast* coherence protocol. In this case, C_2 must recognize that the update is to a data item of which it has a copy, and accept the new value.

Another alternative is that C_1 may broadcast a signal on the bus that any cached copies of L must be invalidated. This is an example of a *write-invalidate* protocol. In this case, C_2 must recognize the signal and invalidate its copy. Subsequently, since C_1

has the only valid copy of L′, any further modifications of L′ by P_1 may be confined to C_1. However, C_1's snoop must monitor any bus read transactions which refer to L′, and intervene to supply the most recently written copy. It may do this either by supplying the copy directly to the requesting cache, or by writing the copy back to memory before allowing the read to proceed.

2.2 Survey of Cache Coherence Protocols

2.2.1 Protocol Terminology

Cache coherence protocols for bus connected shared memory architectures can be grouped into two classes: *write-invalidate* and *write-broadcast* protocols. Write-invalidate protocols maintain coherence by allowing at most one cache in the system to have a dirty copy of a line. When a cache needs to modify a line, it sends an invalidate signal on the bus to cause all other caches to invalidate their copies of the line. Write-broadcast protocols, on the other hand, allow multiple caches to have dirty copies of a line. When any cache modifies the line, it broadcasts the modification so that other caches can update their copies.

This section describes a number of protocols that have been published in the literature. These protocols all have a number of aspects in common, which allow them to be described and compared within a uniform framework. Firstly, they all operate by storing additional status bits (as well as the tag) with each line in a cache. These bits define the line's coherence state with respect to that cache. Secondly, the coherence state is modified in response to processor transactions and bus transactions performed on the line. Depending on the coherence state and the transaction type, the cache may perform different actions to maintain coherence with other caches.

Bus transactions are monitored by special hardware in the cache, often called a *snoop*. This hardware senses the bus control and address signals, and checks the cache tags and status bits at the start of each bus transaction. It modifies the status bits and participates in the transaction as required by the coherence protocol.

A convenient way of describing the coherence protocols is using a state transition diagram that represents an individual cache's actions. The states in the diagram represent the coherence state with respect to the cache for an arbitrary line in the address space. There is a transition from each state for each processor transaction type and each bus transaction type, indicating the new coherence state and any action required of the cache. To clarify the descriptions of the protocols, a common terminology for all protocols is adopted in this thesis.

Firstly, to describe the coherence state of a line in the address space with respect to some cache, three boolean attributes are used: *valid*, *exclusive* and *owned*. If a line is valid with respect to a cache, that cache holds a copy of the line. If a line is exclusive with respect to a cache, there are no other caches that also hold a copy. If a line is owned with respect to a cache, the line is dirty in that cache, and the cache is responsible for updating shared memory at some stage, or passing ownership to some other cache.

Note that if a line is valid with respect to a cache, any of the four combinations of exclusive/not-exclusive and owned/not-owned are feasible. If a line is not valid, the exclusive and owned attributes are not applicable. Thus there are five feasible coherence states. These correspond to the five states in the model proposed in the draft IEEE P896.2 Futurebus cache coherence specification [32, 50]. The correspondence is:

- M: valid/exclusive/owned,
- O: valid/not-exclusive/owned,
- E: valid/exclusive/not-owned,
- S: valid/not-exclusive/not-owned, and
- I: not-valid.

Four of these states, M, E, S and I, are used in a number of commercially implemented "MESI" coherence protocols. The remaining state, O, is less commonly used.

A coherence protocol maintains coherence by ensuring that each line has the correct set of coherence states with respect to each cache in the system. In this thesis, the term *configuration* is used to refer to this set. Different protocols have different allowed configurations, but they all ensure that the most recently written value of a line is accessed by each cache.

The second aspect of a cache coherence protocol that needs to be specified is the set of transactions on a line that may be requested by a processor. The terminology used in this thesis is:

- *read*: read from the line,
- *write*: write to the line,
- *flush*: flush the line from the cache (either when the line is to be replaced on a cache miss, or when shared memory is to be synchronized with the cache).

The last of these transactions, flush, simply refers to the request to remove a line form the cache. Whether memory is updated or other caches are invalidated depends on the particular coherence protocol.

The third aspect is the set of bus transactions that may be initiated by a cache. The terminology used in this thesis is:

- *read-shared*: read from a line that will not be modified in the cache, allowing other caches to keep a copy of the line,
- *read-invalidate*: read of a whole line that may be modified in the cache, requiring other caches to invalidate any copies of the line,

- *invalidate*: an address-only transaction signalling that other caches must invalidate any copies of the line,
- *write-invalidate*: write to a line, with the side-effect that other caches must invalidate any copies of the line,
- *write-update-clean*: broadcast write to a line, updating copies in other caches and in shared memory,
- *write-update-dirty*: broadcast write to a line, updating copies in other caches, but not in shared memory,
- *write-back*: write of a dirty line back to shared memory.

In addition, when a cache detects a read bus transaction to a line for which it has the owned attribute set, the cache must take action to supply the data in place of the shared memory. The term *intervention* is used for this action. The cache may take advantage of the line being transferred on the bus to update shared memory as a side-effect. In this case, the term *reflection* is used. From the point of view of the shared memory, if a cache reflects during a bus read transaction, the read is turned into a write.

2.2.2 Goodman's Write-once Protocol

The write-once protocol was the first published cache coherence protocol specifically for bus connected shared memory multiprocessors [25]. It was developed for implementation with Multibus as the interconnect between caches and memory. Multibus has very limited support for transactions required to support coherence, so the write-once protocol is fairly simple.

The write-once protocol is a write-invalidate copy-back protocol. The coherence states used in write-once are:

- *invalid*: not-valid,
- *valid*: valid/not-exclusive/not-owned,

- *reserved*: valid/exclusive/not-owned, and
- *dirty*: valid/exclusive/owned.

Goodman describes these states as having the following significance. A line is invalid if the cache does not have a copy. A line is valid if the cache has a copy which it has not yet modified. A line is reserved if it has been modified exactly once since being in the valid state, and the modification has been written through to shared memory. Only one cache in the system can have a given line in the reserved state. A line is dirty if it has been modified locally since being in the reserved state, and the modification has not been written through to shared memory.

The bus transactions used in the write-once protocol are:

- read-shared,
- read-invalidate,
- write-invalidate, and
- write-back (although this need not be a separate transaction type from write-invalidate for this protocol).

Goodman's original proposal did not explicitly identify all of these transaction types, but they are included here for uniformity with the descriptions of other protocols. Goodman did not differentiate between the two forms of write, since the protocol does not need to distinguish them. He did not mention read-invalidate, since he did not address the handling of write misses in the protocol. However, Archibald and Baer, in their description of the protocol [2], rectify this omission, and it is their description that is presented below.

The write-once protocol also requires caches to intervene and to reflect. Goodman suggests that reflection can be achieved by writing back to shared memory as a separate action immediately after intervention [25]. This is required on Multibus, since that bus does not support reflection directly.

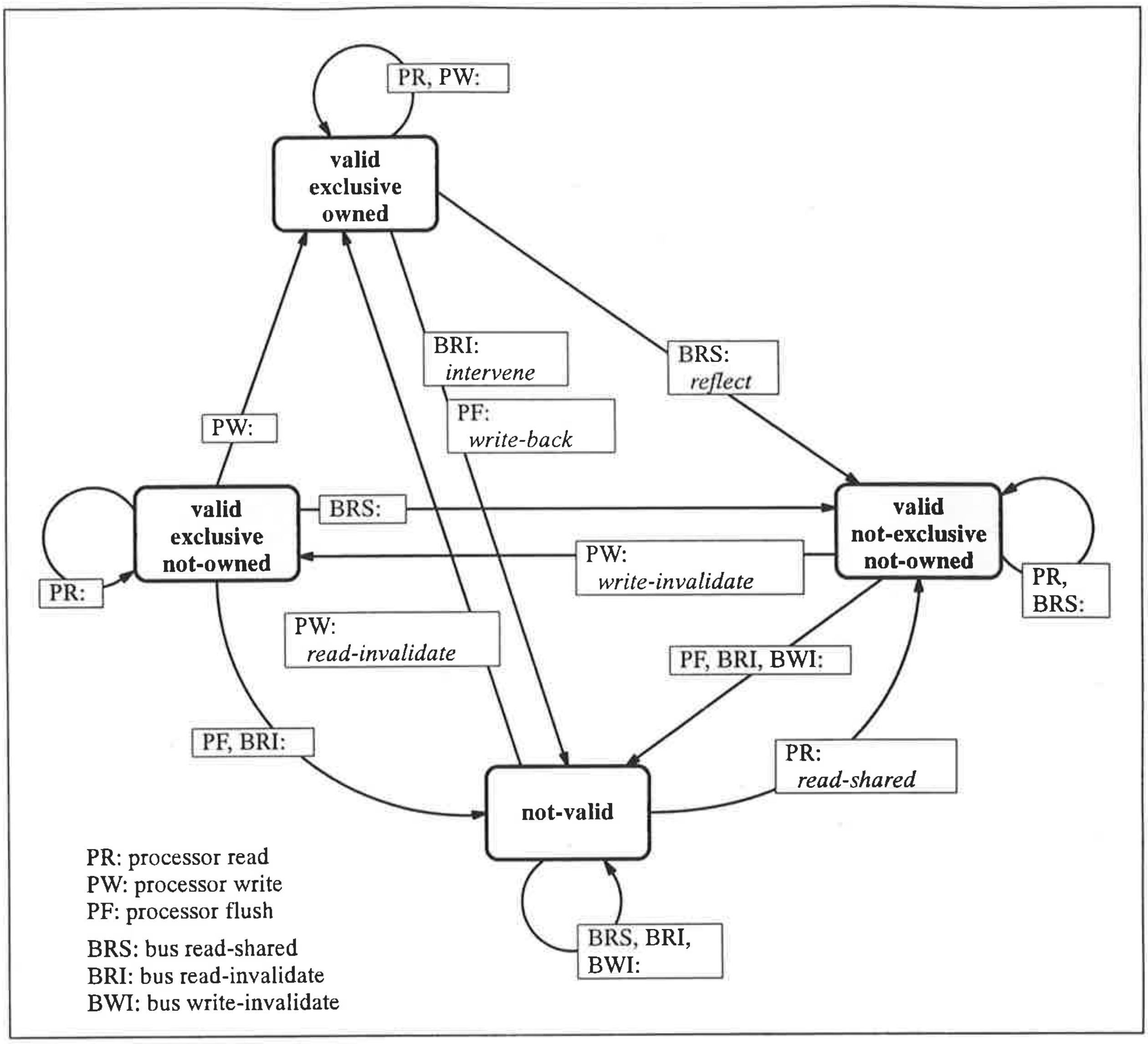


Figure 2-2. The write-once protocol.

The state transition diagram for write-once is shown in Figure 2-2. A line is initially not-valid with respect to all caches. On a read miss, the cache performs a read-shared transaction to fetch the line, and sets the line attributes to valid/not-exclusive/not-owned. If some other cache has a valid/exclusive/owned copy of the line, it responds to the read-shared transaction by reflecting (thus updating shared memory), and changing its attributes to valid/not-exclusive/not-owned. If some other cache has a valid/exclusive/not-owned copy, it clears the exclusive attribute. Hence the result of the transaction is that all valid copies of the line in caches are in the valid/not-exclusive/not-owned state, and are consistent with shared memory.

A read hit occurs when the processor does a read at a line which has the valid attribute set. In these cases, the cache can satisfy the read locally without changing the coherence state of the line.

On a write miss, the cache fetches the line by performing a read-invalidate bus transaction, performing the write locally, and setting the line attributes to valid/exclusive/owned. If some other cache has a valid/exclusive/owned copy of the line, it responds to the read-invalidate transaction by intervening (without updating shared memory). All caches that have copies of the line (apart form the originator) invalidate it. The result is that the cache performing the read-invalidate has the only copy of the line, and can perform further modifications locally.

On a write hit at a valid/not-exclusive/not-owned line, the cache performs the write locally, and also performs a write-invalidate to write through to shared memory and to invalidate the line in other caches. It then has an exclusive copy of the line, so it sets the exclusive attribute. On a write hit at a valid/exclusive/not-owned line, the cache sets the owned attribute, allowing this and subsequent writes to be performed locally. The cache is then the owner of the line, and is thus responsible either for updating shared memory (with a write-back transaction when the line is flushed by the processor, or by reflecting when a bus read-shared transaction occurs), or for passing ownership to another cache (by intervening when a bus read-invalidate transaction occurs).

There are four possible configurations for each line in the write-once protocol. They are:

1. not-valid in all caches,
2. valid/not-exclusive/not-owned in one or more caches,
3. valid/exclusive/not-owned in exactly one cache, and

4. valid/exclusive/owned in exactly one cache.

A line of memory changes between these configurations based on its use by tasks running on processors. The premise behind the protocol is that it provides the best properties of write-through and write-back. If a line is only modified once before being replaced, write-through is more appropriate. On the other hand, if it is modified more than once, write-back reduces the bus usage. Write-once effectively performs write-through on the transition from the valid/not-exclusive/not-owned state to the valid/exclusive/not-owned state, and write-back once the line has the owned attribute.

2.2.3 The Illinois Protocol

The Illinois cache coherence protocol [39] was developed by Papamarcos and Patel at the University of Illinois. It is similar to write-once, with some optimizations included to reduce bus traffic. The most significant of these is that the protocol determines, when a read-miss is handled, whether other caches share a copy of the fetched block. If not, then subsequent modification of the block can proceed without further bus traffic.

Like the write-once protocol, the Illinois protocol is a write-invalidate protocol. The coherence states used are:

- *invalid*: not-valid,
- *exclusive-unmodified*: valid/exclusive/not-owned,
- *shared-unmodified*: valid/not-exclusive/not-owned, and
- *exclusive-modified*: valid/exclusive/owned.

A line is invalid with respect to a cache if the cache does not have a copy. A line is in the exclusive-unmodified state if the cache is the only one with a copy, and the copy is consistent with shared memory. If a line is in the shared-unmodified state, other caches may have copies of the line, all of which are consistent with share memory. Finally, if a line is in the exclusive-modified state, no other cache has a copy of the line, and the line has been modified locally without having been written back yet. These states correspond exactly to the states used in write-once, but the way a line changes between them differs.

The bus transactions used in the Illinois protocol are:

- read-shared,
- read-invalidate,
- invalidate, and
- write-back.

The optimization over write-once shown here is that the write-invalidate transaction is replaced with an address-only invalidate transaction. This allows a cache to gain ownership of a shared line without having to transfer any data on the bus, thus reducing bus usage. The authors do not explicitly identify the write-back transaction [39], but, as with the description of the write-once protocol, write-back is included here for uniformity.

The Illinois protocol requires that caches be able to intervene and to reflect. In [39], Papamarcos and Patel explicitly identify reflection as a mechanism to be provided in the coherency protocol (although not by the name "reflection"), compared to Goodman's suggestion that it be implemented as a separate bus transaction following intervention. This indicates that Papamarcos and Patel envisioned a new bus design for their protocol, since no available buses at the time had any support for reflection.

The Illinois protocol also requires that when a cache issues a read transaction on the bus, all caches with a copy of the line should respond to supply the data [39]. Where more than one cache responds, some form of resolution scheme should be employed to select one uniquely. Furthermore, a requesting cache must be able to determine whether a cache or shared memory responds to its read request, indicating whether the requesting cache gets an exclusive or shared copy of the line. (This further supports the view that the designers envisaged a new bus design for their protocol.)

The premise behind having all caches respond to read requests is that they can respond faster than shared memory, so bus utilization is reduced by having them do so. However, when practical design considerations are taken into account, this may not be the case. For a cache to respond, it must contend for access to the cache data RAM with its client processor, and this additional contention may reduce the performance of the processors and delay cache response. So far as cache coherency is concerned, it does not matter whether some other cache or shared memory supplies the data, provided the requesting cache is able to determine whether other caches have a copy of the line. This can be signalled by the other caches without introducing contention, since it only involves a tag access, and snoop-based cache implementations often have duplicated tag stores to avoid such contention. As can be seen in Section 2.4, the IEEE Futurebus proposal included a signal for this purpose.

The state transition diagram for the Illinois protocol is shown in Figure 2-3. A line is initially not-valid with respect to all caches. On a read miss, a cache issues a read-shared bus transaction to fetch the line. If no other caches hold a copy of the line, the requesting cache sets the line attributes to valid/exclusive/not-owned, since it is the only cached copy in the system. On the other hand, if there are other caches with a copy, the requesting cache sets the line attributes to valid/not-exclusive/not-owned. If another cache has a valid/exclusive/not-owned copy, it responds to the read-shared transaction by clearing the exclusive attribute. A cache with a valid/exclusive/owned copy must reflect to supply the data and coincidentally update shared memory, and change the attributes for its copy to valid/not-exclusive/not-owned. The end result is that all caches with a copy of the line have it in the valid/not-exclusive/not-owned state.

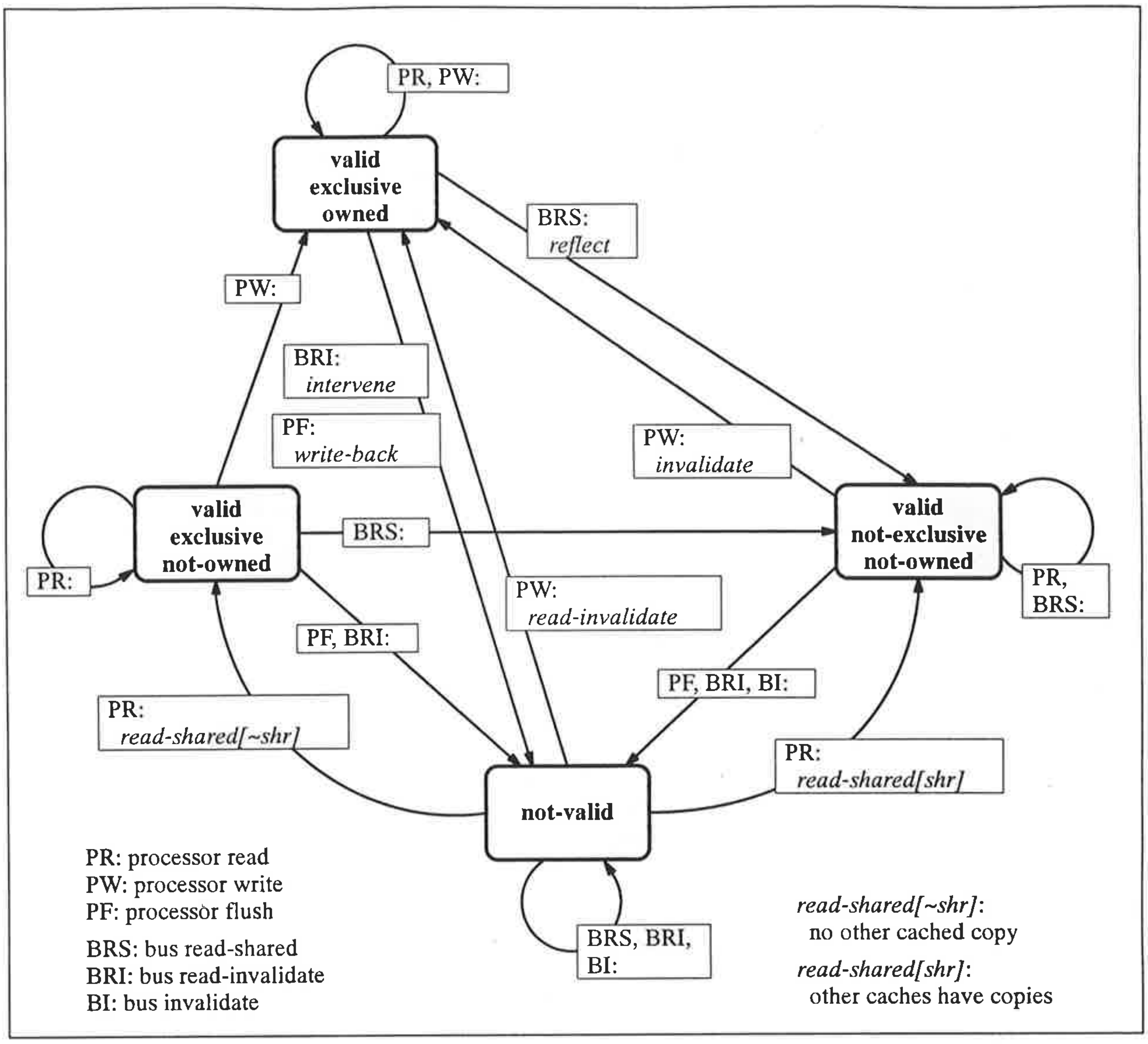


Figure 2-3. The Illinois protocol.

A read hit occurs when the processor does a read at a line which has the valid attribute set. In these cases, the cache can satisfy the read locally without changing the coherence state of the line.

On a write miss, the cache fetches the line by issuing a read-invalidate bus transaction, performing the write locally, and setting the attributes of the line to valid/exclusive/owned. Other caches with a copy of the line (if any) respond to the read-invalidate by intervening and then invalidating their copies. The result is that the requesting cache has the only copy of the line, and can proceed with further modifications locally.

On a write hit at a valid/not-exclusive/not-owned line, a cache issues an invalidate bus transaction. This causes any other caches with a copy of the line to invalidate it. The originating cache changes the line attributes to valid/exclusive/owned, allowing this and subsequent writes to be performed locally. A write hit at a valid/exclusive/not-owned line causes the cache to set the owned attribute and perform the write. Since it is known that no other caches have a copy, no bus transaction is required.

A write back transaction can only occur when a processor flushes a valid/exclusive/owned line. In this case, no other cache should have a copy of the line, so the state diagram does not need to specify a snoop's action for write-back transaction.

There are four possible configurations for each line in the Illinois protocol. They are:

1. not-valid in all caches,
2. valid/not-exclusive/not-owned in one or more caches,
3. valid/exclusive/not-owned in exactly one cache, and
4. valid/exclusive/owned in exactly one cache.

These correspond exactly to the four configurations in the write-once protocol. The main difference is that a line can go directly from the second to the fourth of these configurations, bypassing the third. Whereas write-once effectively uses write-through for the first write to a line, and copy-back for subsequent writes, the Illinois protocol uses copy-back immediately, thus reducing bus usage. One would expect this to yield better performance for writable data that is local to a process, since the number of bus cycles consumed in accessing that data is reduced.

2.2.4 The Synapse Protocol

The Synapse N+1 system, described by Frank, was one of the first commercial shared memory multiprocessors that implemented a cache coherence protocol [21]. The system was targeted at high performance fault tolerant computing applications, and to this end, the architecture included dual buses with a split transaction protocol.

The Synapse cache coherence protocol is a simple write-invalidate copy-back protocol. The coherence states used are:

- *invalid*: not-valid,
- *public*: valid/not-exclusive/not-owned, and
- *private*: valid/exclusive/owned.

A line is invalid if the cache does not have a copy. Frank refers to the distinction between public and private as the *usage* of the line. Public usage means the line is read-only, may be resident in more than one cache, and may include read-only shared data. Private usage means the line includes writable data, so only one cache in the system may have a copy, and this copy may be inconsistent with shared memory. Frank introduced the idea of *ownership*: the cache that has the copy of a line with private usage is the *owner* of the line. If a line has public usage, the shared memory is considered to be the owner. This is consistent with the descriptive framework used in this thesis to describe cache coherence protocols.

Frank describes the implementation of the Synapse cache as using three status bits: valid, usage and modified [21]. As will become clear from the following description, the usage and modified bits should always be in the same state. It is not clear from Frank's description why two separate bits are used, though one might surmise that the modified bit would remain clear if a write operation to a private line fails.

The bus transactions used in the Synapse protocol are:

- read-shared,
- read-invalidate, and
- write-back.

The system also uses the write-invalidate transaction for I/O processors to do DMA writes. This causes all caches with a copy of a line to invalidate it. Because it is only used for I/O operations, it is not included here as part of the cache coherence protocol (following the approaches of other descriptions of this protocol).

The Synapse protocol also requires caches to intervene and reflect. Intervention is achieved by having the owner cache respond to a request for a line. The shared memory knows not to respond, by virtue of having a usage mode bit with each line of memory. If the line is private, some cache is owner, and will intervene. Reflection is achieved by the owner cache returning a busy status to a request. The owner then performs a write-back transaction to shared memory, which will then respond when the original request is retried. Equivalent semantics can be achieved in one transaction on a bus that properly supports reflection, so in this description of the protocol, reflection is not subdivided into its component sub-transactions.

The state transition diagram for the Synapse protocol is shown in Figure 2-4. A line is initially not-valid with respect to all caches. On a read miss, the cache performs a read-shared transaction to fetch the line, and sets the attributes of the line to valid/not-exclusive/not-owned. If the line is in private usage, the cache with the valid/exclusive/owned copy of the line responds to the read-shared transaction by reflecting and changing to the not-valid state. (Recall that this is actually done on the Synapse system by writing back to shared memory, and then letting the shared memory respond.) If the line is in public usage, shared memory responds to the transaction, and any other caches with a copy of the line retain it in the valid/not-exclusive/not-owned state.

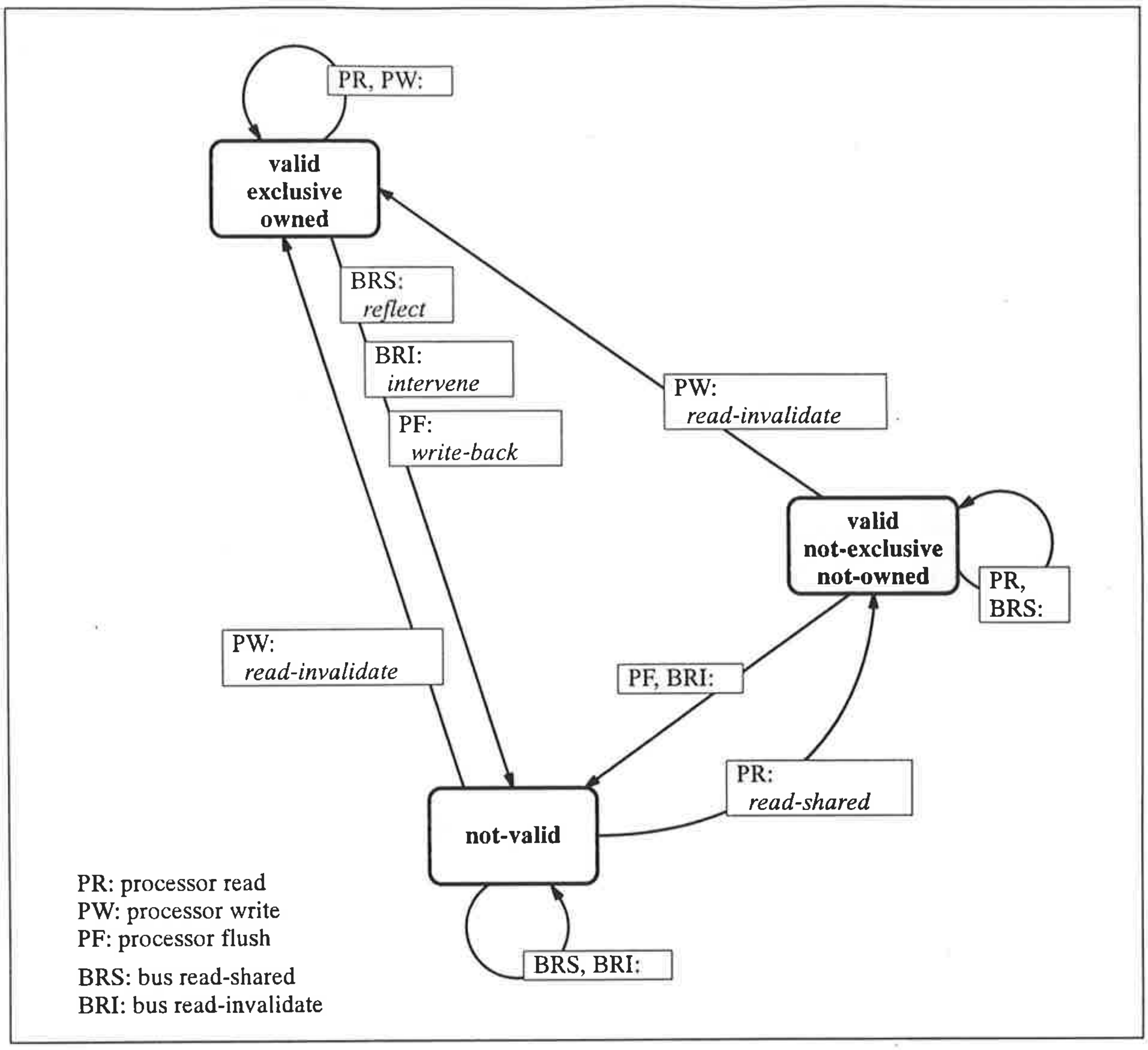


Figure 2-4. The Synapse protocol.

Hence the result of the transaction is that all copies of the line in caches are in the valid/not-exclusive/not-owned state, and are consistent with shared memory.

A read hit occurs when the processor does a read at a line which has the valid attribute set. The cache can satisfy the read locally without changing the coherence state of the line.

On a write miss, the cache fetches the line by performing a read-invalidate bus transaction, performs the write locally, and sets the line attributes to valid/exclusive/owned.

If the line is in private usage, the cache with the valid/exclusive/owned copy of the line responds to the read-invalidate transaction by intervening (without updating shared memory) and then invalidating its copy. If the line is in public usage, any cache with a copy of the line invalidates it. The result is that the cache performing the read-invalidate has the only copy of the line, and can perform further modifications locally; that is, the line is now in private usage.

On a write hit at a valid/not-exclusive/not-owned line, the cache performs a read-invalidate transaction to invalidate the line in other caches, sets the line attributes to valid/exclusive/owned, then performs the write locally. On a write hit at a valid/exclusive/owned line, the writes is performed locally, since no other cache has a copy of the line. When the owned attribute is set, the cache is the owner of the line, and is thus responsible either for updating shared memory (with a write-back transaction when the line is flushed by the processor, or by reflecting when a bus read-shared transaction occurs), or for passing ownership to another cache (by intervening when a bus read-invalidate transaction occurs).

The apparent simplicity of the Synapse protocol is achieved at the cost of some inefficiencies. Firstly, when a write-hit occurs at a valid/not-exclusive/not-owned line, the cache re-reads the line with a read-invalidate transaction, even though it already has a coherent copy of the line. This is because the bus protocol does not include an address-only invalidate transaction to notify other caches that they must invalidate the line. This results in extra bus traffic. Secondly, when a read-miss occurs to a line with private usage, the owning cache reflects and invalidates its copy. If it then needs to read from the line, it suffers a miss, and re-reads the line from shared memory. It is not clear why the owner does not keep a copy of the line in the valid/not-exclusive/not-owned state. There should be little (if any) penalty, and probably some performance gain.

There are three possible configurations for each line in the Synapse protocol. They are:

1. not-valid in all caches,
2. valid/not-exclusive/not-owned in one or more caches, and
3. valid/exclusive/owned in exactly one cache.

The last two configurations correspond to what Frank calls public and private usage of a line. The premise behind the protocol is that it "optimizes system performance by allowing efficient sharing of data while minimizing the overheads of maintaining coherence" ([21], p. 166). While it is true that the protocol's simplicity reduces the complexity of its implementation, the question of the scheme's efficiency compared to other protocols is debatable, and is one of the main issues addressed in this thesis.

2.2.5 The Berkeley Ownership Protocol

The Berkeley cache coherence protocol [35] was designed to improve performance without adding significant cost to a system. The designers note the following constraints on their protocol ([35], p. 277):

> (1) minimize the number of additional bus actions needed to maintain consistency, thus making data sharing reasonably cheap, (2) avoid memory system design, so that commercially available memory boards could be used, and (3) avoid backplane design, although additional signals could be added to an existing backplane and bus protocol to support special communications among the caches.

This is in contrast to some other protocols to be examined in the rest of this section that involved extensive design of new bus backplanes and signalling protocols. The Berkeley protocol, on the other hand, was first implemented in a Multibus system with the addition of one extra control wire. Subsequently, the protocol was incorporated into the SPUR multiprocessor [27] using a modified Nubus.

The Berkeley protocol is a write-invalidate copy-back protocol based on the idea of ownership introduced in the Synapse protocol. The coherence states used are:

- *invalid*: not-valid,
- *unowned*: valid/not-exclusive/not-owned
- *owned exclusive*; valid/exclusive/owned, and
- *owned non-exclusive*: valid/not-exclusive/owned.

A line is invalid if the cache does not have a copy. The unowned state corresponds to the public state in the Synapse protocol, and indicates that the line is present in the cache and may only be read. Other caches may also have copies of the line. The owned exclusive state corresponds to the private state in the Synapse protocol. It indicates that the line has been modified locally, that it is the only cached copy, and that shared memory has not been updated. The owned non-exclusive state is an extension of the set of Synapse states. It is similar to the owned exclusive state, but does allows other caches to hold copies of the line. If any modifications are to be made to the line, these other caches must be informed.

The bus transactions used in the Berkeley protocol are:

- read-shared,
- read-invalidate,
- invalidate, and
- write-back.

The designers also describe the use of write-invalidate transactions [35]; however these are only issued by I/O devices and processors without caches, so they are not include in this description.

The Berkeley protocol requires a cache to intervene when a line it owns is read by some other cache. Shared memory is not updated in this case, and ownership remains with the intervening cache in the case of a read-shared transaction, or is transferred to the reading cache in the case of a read-invalidate transaction. This means that data

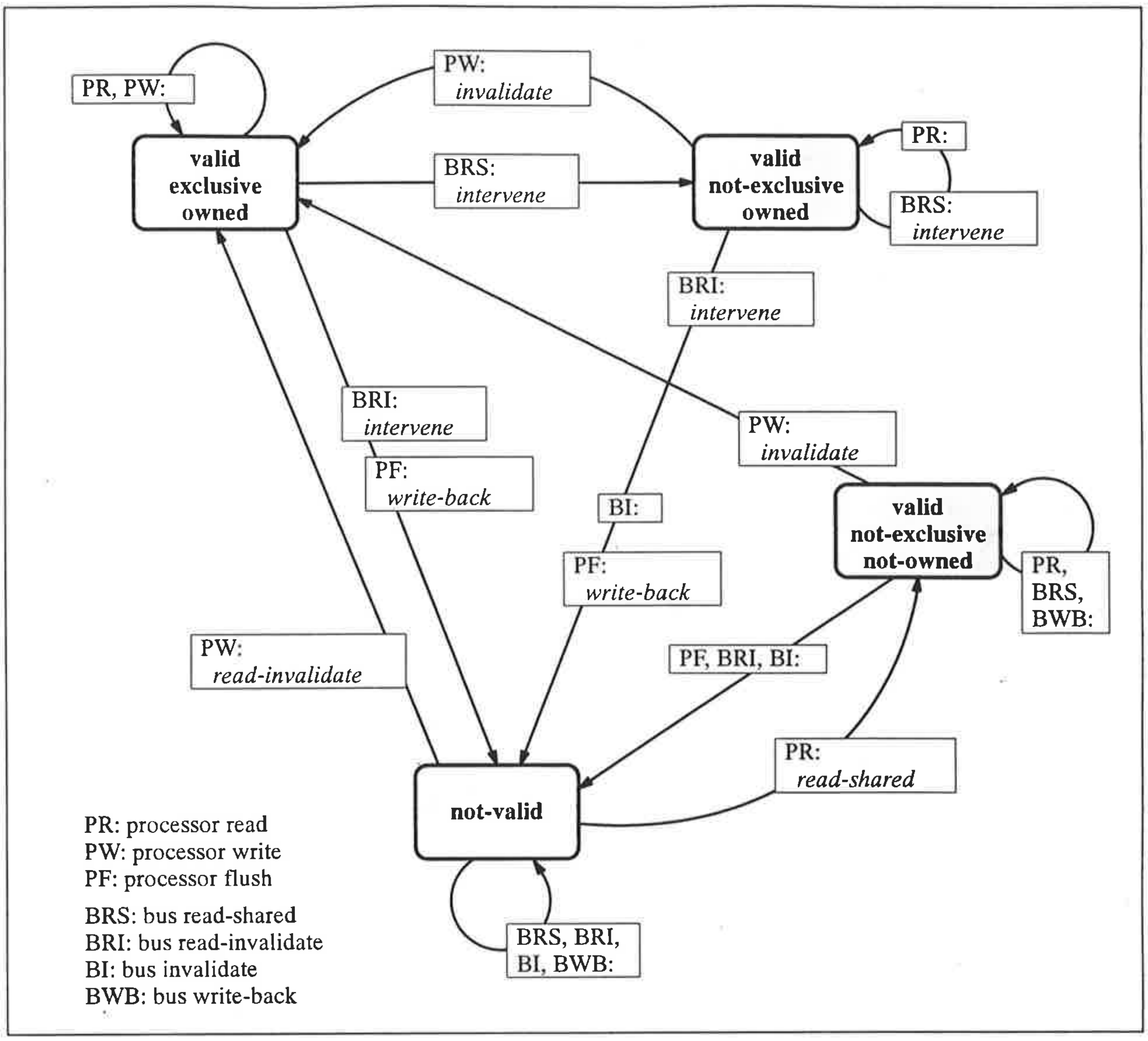


Figure 2-5. The Berkeley protocol.

that is read/write shared is passed amongst the caches, and main memory is not updated until the owner must replace the line.

The state transition diagram for the Berkeley protocol is shown in Figure 2-5. A line is initially not-valid with respect to all caches. On a read miss, the cache performs a read-shared bus transaction to fetch the line, and sets the attributes of the line to valid/not-exclusive/not-owned. If some other cache has the owned attribute set for the line, it intervenes on the read-shared transaction to supply the most up-to-date copy, and sets its attributes to valid/not-exclusive/owned. If the memory is the owner, it services the read-shared transaction. Hence the result of the transaction is that all caches with a copy of the line have it with the exclusive attribute cleared, even though the cached copies may not be consistent with shared memory.

A read hit occurs when the processor does a read at a line which has the valid attribute set. In these cases, the cache can satisfy the read locally without changing the coherence state of the line.

On a write miss, the cache fetches the line by performing a read-invalidate bus transaction to fetch the line, sets its attributes to valid/exclusive/owned, and proceeds with the write locally. The bus transaction causes the current owner of the line to supply it, either by intervening if a cache is the owner, or by simply responding to the read if the shared memory is the owner. In addition, any other cache which has a copy of the line invalidates it. The result is that the cache which performed the read-invalidate becomes owner and has the only valid copy of the line, and can make further modifications locally.

The actions on a write hit depend on the attributes of the line. If the line has the exclusive attribute cleared, there may be other caches in the system with a copy. In this case, the modifying cache issues an invalidate bus transaction to cause other caches to invalidate the line. The modifying cache sets the attributes of its copy of the line to valid/exclusive/owned, and proceeds with the write locally. If, on the other hand, the line is valid/exclusive/owned, then no other cache has a copy of the line, so the modifying cache leaves the line in this state and proceeds with the write locally. In both cases, the result is that the modifying cache becomes owner and has the only valid copy of the line, and can make further modifications locally.

The remaining transitions deal with the case of a line being flushed from the cache by a processor. If the cache is the owner of the line, it is responsible for ensuring that shared memory is updated, and so it issues a write-back bus transaction. Since only one cache can be owner of a line, a bus snoop should never observe a write-back transaction for a line it owns. It may, however, observe a write-back of a valid/not-exclusive/not-owned line, but no action is required in this case, as the owner is simply passing ownership back to the shared memory.

The possible configurations for each line in the Berkeley protocol are:

1. not-valid in all caches,
2. valid/not-exclusive/not-owned in one or more caches,
3. valid/exclusive/owned in exactly one cache, and
4. valid/not-exclusive/owned in exactly one cache and valid/not-exclusive/not-owned in zero or more other caches.

The second and fourth of these correspond to public usage in the Synapse protocol, and the third to private usage. The difference between the two configurations of public usage arises from the Berkeley protocol designers' choice of avoiding writing a line back to shared memory when usage changes from private to public. Under the Berkeley protocol, data that is read-only or read shared will be in the second configuration. Data that is local to a process will be in the second or third configuration. Data that is read/write shared will alternate between the third and fourth configurations, with no additional shared memory access required until the line is flushed by the owner cache. Compare this with the write-once protocol, in which a shared memory write is required whenever a cache modifies possibly shared data. The Berkeley protocol designers show that the write-once protocol issues significantly more bus traffic than their protocol [35].

2.2.6 The Sun MBus Protocol

In 1991 Sun Microsystems announced the first of its multiprocessor systems, the SPARCsystem 600MP [47], a two- or four-processor system designed for network server applications. They have since introduced a number of desktop multiprocessor workstations, using the same multiprocessor technology. These systems are based on small CPU/cache/MMU modules, interconnected with shared memory using a bus called MBus, which implements the cache coherence protocol described in this section.

The MBus protocol is a write-invalidate copy-back protocol, which can be viewed as an extension of the Berkeley protocol. The coherence states used are:

- *invalid*: not-valid,
- *exclusive clean*: valid/exclusive/not-owned,
- *shared clean*: valid/not-exclusive/not-owned,
- *exclusive modified*; valid/exclusive/owned, and
- *shared modified*: valid/not-exclusive/owned.

A line is invalid if the cache does not have a copy. The exclusive clean state indicates that the cache is the only cache with a copy, and that it is consistent with shared memory. This corresponds to the exclusive-unmodified state in the Illinois protocol, and is an extension over the Berkeley protocol. The shared clean state corresponds to the unowned state in the Berkeley protocol, and indicates that the line is present in the cache and may only be read. Other caches may also have copies of the line. The exclusive modified state corresponds to the owned exclusive state in the Berkeley protocol. It indicates that the line has been modified locally, that it is the only cached copy, and that shared memory has not been updated. Finally, the shared modified state corresponds to the owned non-exclusive state in the Berkeley protocol. It is similar to the exclusive modified state, but does allow other caches to hold copies of the line. If any modifications are to be made to the line, these other caches must be informed.

The bus transactions used in the MBus protocol are:

- read-shared,
- read-invalidate
- invalidate, and
- write-back.

The designers note that DMA I/O transactions also participate in the cache coherence protocol using write-invalidate transactions [47]. This is not includes in the description of the protocol in this thesis.

Like the Berkeley protocol, the MBus protocol requires a cache to intervene when a line it owns is read by some other cache. Shared memory is not updated in this case, and ownership remains with the intervening cache in the case of a read-shared transaction, or is transferred to the reading cache in the case of a read-invalidate transaction. The extension to the Berkeley protocol that allows use of the extra coherence state (valid/not-shared/not-owned) is the inclusion of a *sharing* status signal on the bus, as in the Illinois protocol. This is used on a read-shared transaction to allow the requester to determine whether to clear the exclusive attribute when it fetches a line.

The state transition diagram for the MBus protocol is shown in Figure 2-6. A line is initially not-valid with respect to all caches. On a read miss, the cache performs a read-shared bus transaction to fetch the line. If no other caches hold a copy of the line, the requesting cache sets the line attributes to valid/exclusive/not-owned, since it is the only cached copy in the system.

On the other hand, if there are other caches with a copy, the requesting cache sets the line attributes to valid/not-exclusive/not-owned. If some other cache has the owned attribute set for the line, it intervenes on the read-shared transaction to supply the most up-to-date copy, and sets its attributes to valid/not-exclusive/owned. If no other cache is the owner, shared memory services the read-shared transaction, and any other

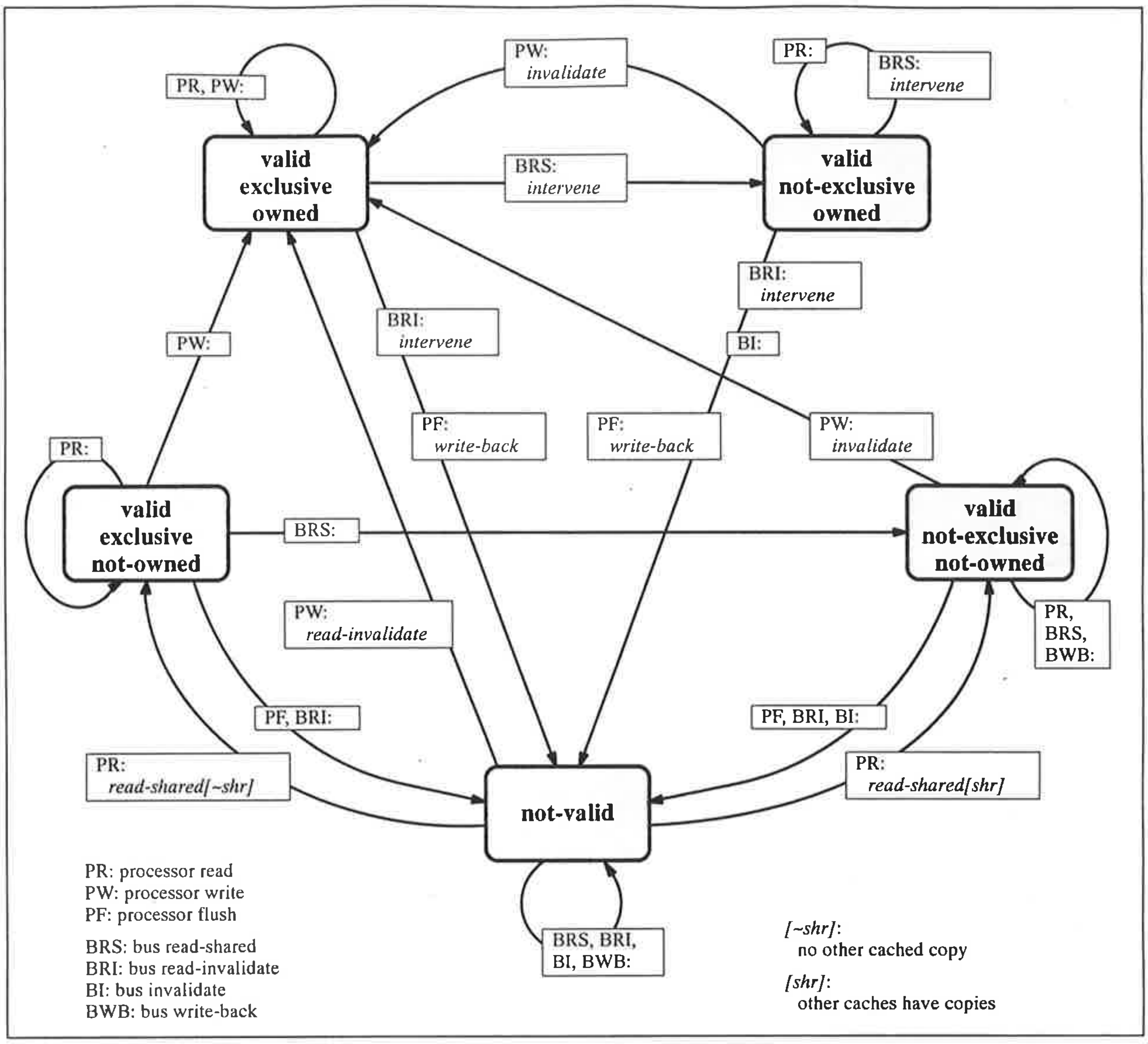


Figure 2-6. The MBus protocol.

cache with a copy of the line sets its attributes to valid/not-exclusive/not-owned. Hence the result of the transaction is that all caches with a copy of the line have it with the exclusive attribute cleared, even though the cached copies may not be consistent with shared memory.

A read hit occurs when the processor does a read at a line which has the valid attribute set. In these cases, the cache can satisfy the read locally without changing the coherence state of the line.

A write miss is handled in the same way as the Berkeley protocol. The cache fetches the line by performing a read-invalidate bus transaction to fetch the line, sets its attributes to valid/exclusive/owned, and proceeds with the write locally. The bus transaction causes the current owner of the line to supply it, either by intervening if a cache is the owner, or by simply responding to the read if the shared memory is the owner. In addition, any other cache which has a copy of the line invalidates it. The result is that the cache which performed the read-invalidate becomes owner and has the only valid copy of the line, and can make further modifications locally.

The actions on a write hit depend on the attributes of the line. If the line has the exclusive attribute cleared, there may be other caches in the system with a copy. In this case, the modifying cache issues an invalidate bus transaction to cause other caches to invalidate the line. The modifying cache sets the attributes of its copy of the line to valid/exclusive/owned, and proceeds with the write locally. If, on the other hand, the exclusive attribute of the line is set, then no other cache has a copy of the line, so the modifying cache sets the owned attribute and proceeds with the write locally. In both cases, the result is that the modifying cache becomes owner and has the only valid copy of the line, and can make further modifications locally.

When the client processor flushes a line that has the owned attribute set, the cache issues a write-back bus transaction. Since only one cache can be owner of a line, a bus snoop should never observe a write-back transaction for a line it owns. As in the Berkeley protocol, it may observe a write-back of a valid/not-exclusive/not-owned line, but no action is required in this case, as the owner is simply passing ownership back to the shared memory.

There are five possible configurations for each line in the MBus protocol. They are:

1. not-valid in all caches,

2. valid/exclusive/not-owned in exactly one cache,

3. valid/exclusive/owned in exactly one cache,
4. valid/not-exclusive/not-owned in one or more caches, and
5. valid/not-exclusive/owned in exactly one cache and valid/not-exclusive/not-owned in zero or more other caches.

A line is in one of the second or third configuration when it contains data which is not currently being shared. Read-only private data remains in the second configuration, whereas data that is modified changes to the third configuration. In the latter case, the cache is the owner of the data, and must update shared memory before evicting the line. Data that is subsequently read shared between processors changes to one of the fourth or fifth when the second cache accesses it. If the first cache does not own the line, it changes to the fourth configuration. If the first cache does own the line, it retains ownership and the line changes to the fifth configuration. Whenever any cache writes to a shared line, all other caches are invalidated, and ownership passes to the writer. The line changes to the second configuration.

The optimization over the Berkeley protocol provided by the inclusion of the valid/exclusive/not-owned state is evident when a read followed by a write to private data is considered. In the Berkeley protocol, the write induces an invalidate bus transaction, since the cache is not able to tell that there are no other cached copies. In the MBus protocol, the cache uses the sharing signal on the read to determine that it should leave the exclusive attribute set, and so can proceed with the write without having to notify other caches. This not only reduces bus traffic, but also reduces contention on other caches between the snoops and the client processors for access to the tag stores.

2.2.7 The Dragon Protocol

The Dragon cache coherence protocol is used in the Dragon multiprocessor developed at the Xerox Palo Alto Research Center [37]. The description presented here is based on that presented by Archibald and Baer [2].

The Dragon protocol is a write-broadcast protocol. The coherence states used are:

- *valid–exclusive*: valid/exclusive/not-owned,
- *shared–clean*: valid/not-exclusive/not-owned,
- *dirty*: valid/exclusive/owned, and
- *shared–dirty*: valid/not-exclusive/owned.

The not-valid state is not explicitly represented in a cache using the Dragon protocol. This is because lines can be write-shared between caches, so invalidations are not necessary. A line is only evicted when replaced on a miss, and the cache is filled on cold start, so a cache entry is never invalid in the Dragon system. However, since this thesis describes the protocol in terms of the state of a line in the address space (rather than a cache entry's state), the not-valid state is included.

The Dragon protocol as described by Archibald and Baer only uses two types of bus transaction: a read transaction of a whole line, and a broadcast write to a single word. These correspond to:

- read-shared, and
- write-update-dirty.

The description does not explicitly mention any form of write-back transaction for flushing a line from a cache to shared memory, however, for completeness, the write-back transaction is included in in this description.

The Dragon protocol requires an owner cache to intervene when it observes a read-shared bus transaction. In addition, like the Illinois system, the Dragon bus includes a *sharing* signal to allow cache snoops to indicate that they have a copy of a line being accessed by a bus transaction.

The state transition diagram for the Dragon protocol is shown in Figure 2-7. A line is initially not-valid with respect to all caches. On a read miss, a cache issues a read-

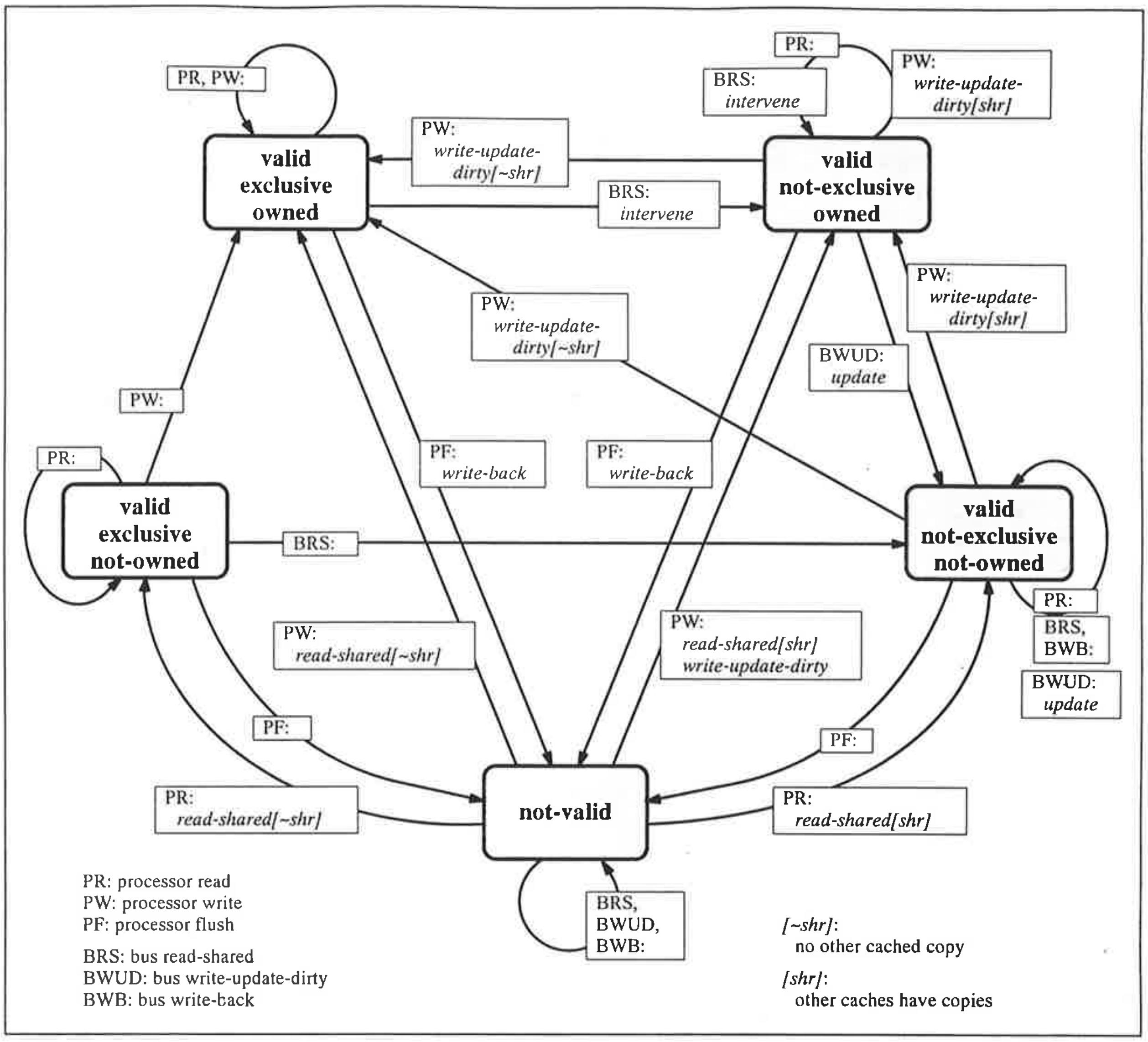


Figure 2-7. The Dragon protocol.

shared bus transaction to fetch the line. If no other caches hold a copy of the line, the sharing signal on the bus is left negated. The requesting cache sets the line attributes to valid/exclusive/not-owned, since it is the only cached copy in the system.

On the other hand, if there are other caches with a copy, they assert the sharing signal, and the requesting cache sets the line attributes to valid/not-exclusive/not-owned. If another cache has a valid/exclusive/not-owned copy, it responds to the read-shared transaction by clearing the exclusive attribute. A cache with an owned copy must intervene to supply the data, and clear the exclusive attribute for its copy. The end result is that all caches with a copy of the line have the exclusive attribute cleared.

A read hit occurs when the processor does a read at a line which has the valid attribute set. In these cases, the cache can satisfy the read locally without changing the coherence state of the line.

On a write miss, the cache fetches the line by performing a read-shared bus transaction. Other caches respond to this transactions exactly as described for a read miss. If the requesting cache detects the sharing signal negated, it can perform the write locally, and sets the line to valid/exclusive/owned. If the sharing signal is asserted, the requesting cache must follow the read-shared with a write-update-dirty in order to maintain coherence with the other caches. The requesting cache sets its copy of the line to valid/not-exclusive/owned, assuming ownership of the line. All other caches set their copies to valid/not-exclusive/not-owned.

On a write hit at a line with the exclusive attribute set, a cache can perform the write locally, and sets the owned attribute. If, however, the exclusive attribute is cleared, the cache issues a write-update-dirty transaction to maintain coherence with other caches, and sets the owned attribute. The other caches accept the new data, and a previous owner clears the owned attribute. The requesting cache also observes the sharing bus signal to determine whether the line is still in fact shared, and clears or sets the exclusive attribute appropriately. (The line may have been evicted by all other caches since the last time sharing status was detected.)

A significant feature of the Dragon protocol is that write broadcasts do not update shared memory. Instead, ownership is passed around to the most recent writer, which must write the line back to shared memory using a write-back transaction when it replaces the line. If the owner has the exclusive attribute cleared when it writes back, the transaction will be sensed by the snoops of other caches with valid/not-exclusive/

not-owned copies of the line. However, they need take no action in response to the transaction.

There are five possible configurations for each line in the Dragon protocol. They are:

1. not-valid in all caches,
2. valid/exclusive/not-owned in exactly one cache,
3. valid/exclusive/owned in exactly one cache,
4. valid/not-exclusive/not-owned in one or more caches, and
5. valid/not-exclusive/owned in exactly one cache and valid/not-exclusive/not-owned in zero or more other caches.

A line is in one of the second or third configurations when it contains data which is not currently being shared. Read-only private data remains in the second configuration, whereas data that is modified changes to the third configuration. In the latter case, the cache is the owner of the data, and must update shared memory before evicting the line. Data that is shared between processors changes to one of the fourth or fifth configuration when the second cache accesses it. If the first cache does not own the line, it changes to the fourth configuration. If the first cache does own the line, it retains ownership as long as the data is only read-shared. (The line is in the fifth configuration.) When one of the caches writes the data, the others are updated, and ownership passes to the writer. (The line stays in the fifth configuration.)

The premise behind the Dragon protocol is that caches are faster than shared memory, so transactions to shared memory should be avoided where possible. For this reason, write-shared data is broadcast amongst caches, without updating shared memory until the line is replaced in the owner. This is related to the assumption in the Illinois protocol that caches can satisfy reads faster than shared memory, and so the same counter arguments involving contention with the client processor within a cache also apply to the Dragon protocol.

Another source of performance loss, identified by the designers of the Firefly protocol (see Section 2.2.9), is that write broadcasts will continue to a shared line for as long as the line remains in other caches, even if their client processors no longer require the data. Compare this with write-invalidate coherence protocols, where a processor write causes the line to become private by invalidating other cached copies.

2.2.8 The Original Firefly Protocol

The Firefly cache coherence protocol is used in the Firefly multiprocessor workstations developed at the Digital Equipment Corporation Systems Research Centre [52]. Archibald and Baer [2] describe an early version of the protocol that contains fewer states than the protocol finally published for Firefly. The original protocol is described here first, and the published protocol is described in the next section.

The original Firefly protocol is similar to the Illinois protocol described in Section 2.2.3, but is write-broadcast instead of write-invalidate. The four coherence states used are:

- not-valid,
- valid/exclusive/not-owned,
- valid/not-exclusive/not-owned, and
- valid/exclusive/owned.

These states correspond exactly to the four states used in the Illinois protocol. As in the Dragon protocol, the not-valid state is not explicitly represented in a cache. Lines can be write-shared between caches, so invalidations are not necessary. A line is only evicted when replaced on a miss, and a special scheme is used to fill a cache on cold start, so a cache entry is never invalid in the original Firefly system. However, the not-valid state is included in this description, as it deals with the coherence state of a line in the address space, not a cache entry.

The bus transactions used in the original Firefly protocol are:

- read-shared,
- write-update-clean, and
- write-back.

In the description of this protocol by Archibald and Baer, the transaction type used to write a line back to shared memory when it is flushed from a cache is not explicitly identified. This discussion uses the write-back transaction type for uniformity.

The original Firefly protocol requires a cache to reflect when it observes a read-shared bus transaction. Like the Illinois protocol, all caches with a copy of the line supply the data. However, instead of selecting just one of them to put the data on the bus, the Firefly system allows all caches to drive the bus, on the premise that they will all drive the same value, since the caches are coherent. In addition, like the Illinois system, the Firefly bus includes a *sharing* signal to allow cache snoops to indicate that they have a hit on a line being accessed by a bus transaction.

The state diagram for the original Firefly protocol is shown in Figure 2-8. The state changes for a line in response to processor read requests and processor flushes are identical to those in the Illinois protocol. However, the state changes in response to a processor write request are quite different, since the protocol is write-broadcast.

A line is initially not-valid with respect to all caches. On a read miss, a cache issues a read-shared bus transaction to fetch the line. If no other caches hold a copy of the line, the requesting cache sets the line attributes to valid/exclusive/not-owned, since it is the only cached copy in the system. On the other hand, if there are other caches with a copy, the requesting cache sets the line attributes to valid/not-exclusive/not-owned. If another cache has a valid/exclusive/not-owned copy, it responds to the read-shared transaction by clearing the exclusive attribute. A cache with a valid/exclusive/owned copy must reflect to supply the data and coincidentally update shared memory, and

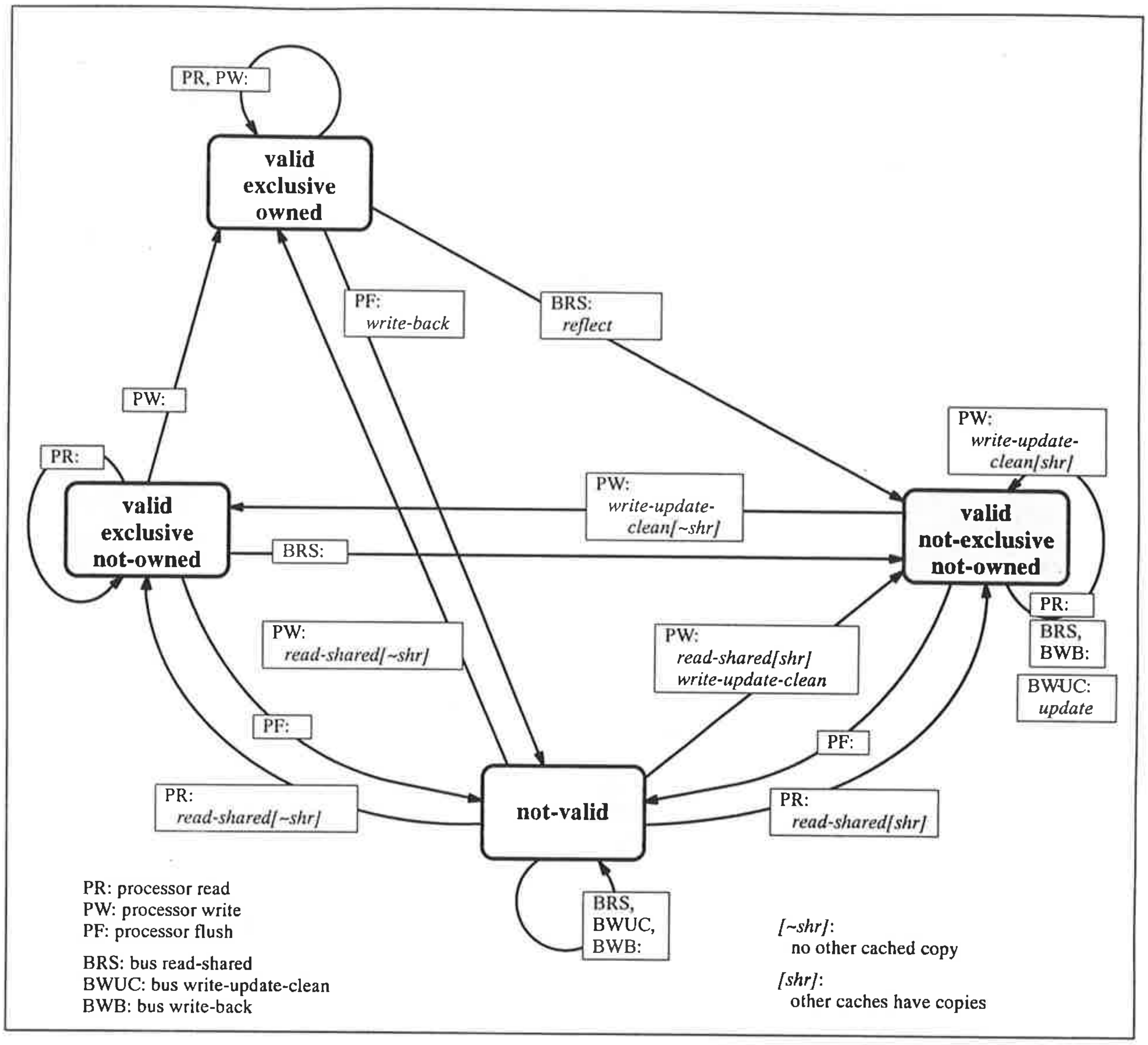


Figure 2-8. The original Firefly protocol.

change the attributes for its copy to valid/not-exclusive/not-owned. The end result is that all caches with a copy of the line have it in the valid/not-exclusive/not-owned state.

A read hit occurs when the processor does a read at a line which has the valid attribute set. In these cases, the cache can satisfy the read locally without changing the coherence state of the line.

On a write miss, the cache firstly fetches the line by issuing a read-shared bus transaction and senses whether other caches in the system have a copy of the line. If no other caches hold a copy, the requesting cache performs the write locally and sets the line attributes to valid/exclusive/owned, since it is the only cache in the system with a copy of the line. On the other hand, if there are other cached copies of the line in the system, the requesting cache follows the read-shared transaction with a write-update-clean transaction to update the other copies and shared memory. The requesting cache also updates its own copy and sets the line attributes to valid/not-exclusive/not-owned.

On a write hit at a valid/not-exclusive/not-owned line the cache issues a write-update-clean transaction to update any other copies that may be held in other caches and to update shared memory, as well as updating its own copy. If there are other cached copies, they are also in the valid/not-exclusive/not-owned state. All of the caches, including the requesting cache, leave the attributes for the line unchanged. On the other hand, if there are no other cached copies, the requesting cache changes the line attributes to valid/exclusive/not-owned. On a write hit at a valid/exclusive/not-owned line the cache sets the owned attribute and performs the write locally. No bus transaction is required, since no other caches have a copy of the line.

As in the Illinois protocol, there are four possible configurations for a line in the original Firefly protocol. They are:

1. not-valid in all caches,
2. valid/not-exclusive/not-owned in one or more caches,
3. valid/exclusive/not-owned in exactly one cache, and
4. valid/exclusive/owned in exactly one cache.

A line that is read-only and is private to one task on a single processor moves between the first and third configurations. When the private line is modified, it moves to the fourth configuration, with updates performed locally. A shared line moves between the first and second configurations, with updates being broadcast to all cached copies. If a line is initially accessed as though it were private, but subsequently becomes shared, it moves from the third or fourth configuration to the second.

2.2.9 The Published Firefly Protocol

The Firefly protocol published in [52] is a write-broadcast protocol, similar in some respects to the Dragon protocol. It effectively uses all five possible coherence states, with all four combinations of not-exclusive/exclusive and not-owned/owned being implemented with two attribute bits, *shared* and *dirty* (owned), for each entry in the cache. The omission of the not-valid state is carried over into the published Firefly protocol. However, the not-valid state is included in this description, as it deals with the coherence state of a line in the address space, not a cache entry.

The published Firefly protocol only uses a single type of read bus transaction and a single type of write bus transaction. They correspond to:

- read-shared, and
- write-update-clean.

No distinction is made between writing a line to update data and writing back to shared memory when a line is flushed from a cache. The designers evidently chose to maintain simplicity of design by reducing the number of types of bus transactions supported.

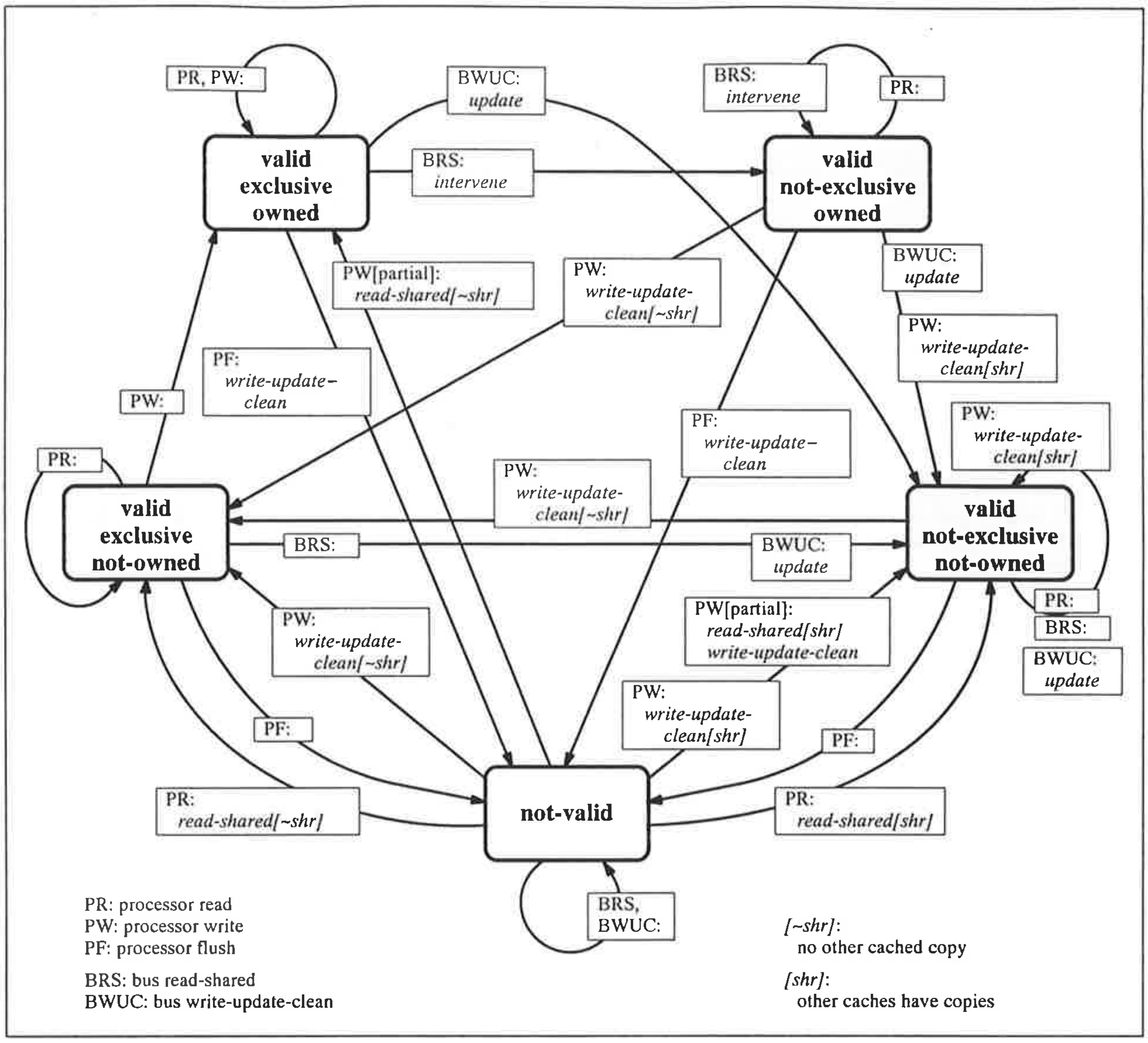
The published Firefly protocol requires a cache to intervene instead of reflect when it observes a read-shared bus transaction. As in the original protocol, all caches with a copy of the line drive the bus to supply the data. Also, the *sharing* signal is retained in the published protocol to allow cache snoops to indicate that they have a hit on a line being accessed by a bus transaction.

One significant difference between the published Firefly system and other systems is that it is uses a cache line size of one word. Since most processor writes are to a whole word, when a dirty line is modified, a write-update-clean transaction to update the word effectively makes the whole line clean again. The published Firefly protocol makes use of this operation to optimize bus traffic for write-shared lines. However, this operation cannot be used for partial word modification (byte or halfword writes), and in these cases, the protocol resorts to a read-shared followed by a write-update-clean to effect the modification.

The state transition diagram for the published Firefly protocol is shown in Figure 2-9. A line is initially not-valid with respect to all caches. On a read miss, a cache issues a read-shared bus transaction to fetch the line, sets the valid attribute, and clears the owned attribute. If other caches have a copy of the line, they intervene to supply the data, assert the sharing signal on the bus, and clear the exclusive attribute for their copy of the line. The requesting cache also clears the exclusive attribute bit in its entry. On the other hand, if no other caches have a copy of the line, the data is supplied by shared memory, the sharing bus signal is left unasserted, and the requesting cache sets the exclusive attribute bit in its entry. When a read hit occurs, the cache can satisfy the read locally without changing the coherence state of the line, and no other cache needs to be notified.

The action on a write miss depends on whether the write is to a whole word (the most frequent case) or to a partial word. In the former case, the cache effectively performs write-through with allocation by issuing a write-update-clean transaction and writing the line in the cache. Other caches with a copy of the line update the data and clear the exclusive attribute in their entries, and assert the sharing signal on the bus. If the requesting cache sees this signal asserted, it sets its copy to valid/not-exclusive/not-owned, otherwise it sets it to valid/exclusive/not-owned.

In the case of a miss on a partial write, the cache issues a read-shared bus transaction to fetch the word. Other caches respond to this transaction exactly as described for a read miss. If the requesting cache detects the sharing signal negated on the bus, it can



PR, PW:
BWUC: update
BRS: intervene
PR:
valid exclusive owned
BRS: intervene
valid not-exclusive owned
BWUC: update
PW[partial]: read-shared[~shr]
PW: write-update-clean[~shr]
PW: write-update-clean[shr]
PF: write-update-clean
PW:
PW: write-update-clean[shr]
PF: write-update-clean
PR:
PW: write-update-clean[~shr]
valid exclusive not-owned
BRS:
BWUC: update
valid not-exclusive not-owned
PR:
BRS:
BWUC: update
PW[partial]: read-shared[shr] write-update-clean
PW: write-update-clean[~shr]
PW: write-update-clean[shr]
PF:
PF:
PR: read-shared[~shr]
not-valid
PR: read-shared[shr]
BRS, BWUC:
PR: processor read
PW: processor write
PF: processor flush
BRS: bus read-shared
BWUC: bus write-update-clean
[~shr]: no other cached copy
[shr]: other caches have copies

Figure 2-9. The Firefly protocol.

perform the partial write locally, and sets the line to valid/exclusive/owned. If the sharing signal is asserted, the requesting cache must follow the read-shared with a write-update-clean in order to maintain coherence with the other caches. In this case, all caches, including the requester, set the line to valid/not-exclusive/not-owned.

On a write hit at a line with the exclusive attribute set, a cache can perform the write locally, and sets the owned attribute. If, however, the exclusive attribute is cleared, the cache issues a write-update-clean transaction to maintain coherence with other caches, and clears the owned attribute. It also observes the sharing bus signal to determine whether the line is still in fact shared, and clears or sets the exclusive attribute appropriately. (The line may have been evicted by all other caches since the last time sharing status was detected.)

When an owned line is to be evicted from a cache on a read or write miss, the cache issues a write-update-clean transaction to update shared memory. The consequence of the protocol not using distinct bus transaction types for write back and normal coherent writes is that other caches that share a copy of the line being written back will still take a copy of the line, even though it is not necessary. The penalty is the small amount of extra contention between the cache snoop and the client processor, but this may be compensated for by the simplicity of having fewer transaction types.

There are five possible configurations for each line in the published Firefly protocol. They are:

1. not-valid in all caches,
2. valid/exclusive/not-owned in exactly one cache,
3. valid/exclusive/owned in exactly one cache,
4. valid/not-exclusive/not-owned in one or more caches, and
5. valid/not-exclusive/owned in exactly one cache and valid/not-exclusive/not-owned in zero or more other caches.

A line is in one of the second or third configurations when it contains data which is not currently being shared. Read-only private data remains in the second configuration, whereas data that is modified changes to the third configuration. In the latter case, the cache is the owner of the data, and must update shared memory before evicting the line. Data that is shared between processors changes to one of the fourth or fifth configuration when the second cache accesses it. If the first cache does not own the line, it changes to the fourth configuration. If the first cache does own the line, it retains ownership as long as the data is only read-shared. (The line is in the fifth configuration.) As soon as any cache writes the data, the write is effectively treated as a write-through operation, and ownership reverts to the shared memory. (The line changes to the fourth configuration.)

The premise behind the published Firefly protocol, stated by the designers, is that a copy-back strategy is appropriate for lines that are not shared, whereas write-through is more appropriate for shared lines. The use of the sharing bus signal allows a cache to determine when sharing ceases, at which time it can revert from write-through to copy-back. The designers note that the disadvantage of this approach is that write-through continues to be used for a shared line for as long as a line remains in other caches, even if their client processors no longer require the data. (This problem also arises with the Dragon protocol, as mentioned in Section 2.2.7.) Compare this with write-invalidate coherence protocols, where a processor write causes the line to become private by invalidating other cached copies, thus allowing a copy-back strategy to be used for the line. The Firefly designers note that write-invalidate protocols perform poorly when write sharing occurs, since an invalidated line containing shared data must be re-fetched when next needed. However, subsequent studies, such as that of Eggers and Katz [18], suggest that in practice, the amount of write-sharing is small, and so this overhead may not be significant.

As mentioned above, the Firefly protocol as published deals with cache lines that are one word in size, allowing some optimizations to reduce bus traffic. It is not clear how best to adapt the protocol to handle cache lines of larger than one word. One possibility is to treat all processor writes as partial line writes, and deal with them according to the state transition rules in Figure 2-9. However, this would require that the whole cache line be broadcast using a *write-update-clean* transaction when a cache has a write hit in response to a client processor write request. The reason for this is that some other cache may be the owner of the line, with several words in the line being dirty. The owner responds to the *write-update-clean* transaction by copying the data and relinquishing ownership. This requires that the caches copies be consistent with shared memory, and the only way to guarantee this is to broadcast the whole line. The problem with this approach is that it would generate excessive bus traffic, transmitting data that, most of the time, is not modified. This is counter to the premise behind the protocol. Hence it is unlikely that the protocol as published would be used for caches with a line size of larger than one word. An alternative is to revert to the earlier version of the protocol described in Section 2.2.8.

2.2.10 The RB and RWB Protocols

The last of the published cache coherence protocols discussed in this thesis is the RWB (read write broadcast) protocol of Rudolph and Segall [44]. This scheme is an extension of the RB (read broadcast) protocol described in the same source. These protocols are rarely cited in the subsequent literature on the subject, for a number of reasons.

The first problem with the RB and RWB protocols is that they are predicated on the assumption that a cache line is one word in size. The authors do not discuss how partial writes should be handled, nor suggest any generalization to larger line sizes. It is clear from the large number of studies analyzing cache performance for various line sizes, that a larger size is desirable for all practical systems.

The second problem is that the protocols rely on cache snoops observing read transactions on the bus, and copying the data into their cache memories. The motivation for this, as stated by the authors, is to optimize performance for reads over writes. A snoop copying data from the bus on a read is optimistically assuming that its client processor will read that data soon, so the act of copying the data potentially avoids an extra bus transaction. However, the bus signaling protocol to synchronize the requester, the responder and an arbitrary number of snoops to permit this kind of transaction is in practice prohibitively complex and expensive. In addition, the protocols require caches to intervene on reads when they have the most up to date copy of a line. The authors suggest that this be done by interrupting the read, writing the data back to shared memory, and then retrying the read. The complexity of allowing snoops to copy the data being read prohibits the optimization of performing a reflecting read as one bus transaction.

Unlike all of the previously discussed protocols, these protocols appear never to have been implemented in a real computer system. Certainly the authors make no mention in their paper of an implementation. Given the problems outlined here, and the fact that the literature gives little further comment on these protocols, they are not described further in this thesis. The interested reader is referred to [44] for a detailed description.

2.2.11 Correctness of Coherence Protocols

It is appropriate at this point to comment on the correctness of the published cache coherence protocols. The object of these protocols is to ensure that each processor observes the most recently written value of any datum when it accesses the memory hierarchy. For the simpler protocols, one can be convinced by inspection that this requirement is met. However, for the more complex protocols, it is not obvious, and some form of verification is desirable.

The framework used in this thesis for description of the protocols can also be used as the basis of their formal verification. The operation of each cache is described in terms of a finite state machine, which makes transitions based on the current state of a line, the action requested by the client processor, and bus transactions observed by the cache snoop. A system consisting of a collection of processor/cache pairs and a shared memory can be modelled as a composite state machine, with the global state based on the collected states of the component caches. In this composite state machine, transitions are initiated solely by requests from client processors.

An implicit assumption inherent in all of the protocols is that global state transitions are atomic. From this assumption comes the requirement that global actions for client processor requests be serialized. Only private operations within a cache, not requiring reference to other cache states, can be performed in parallel. (Fortunately, these are the majority of operations.) Without the assumption of atomicity, the coherence protocols would not operate correctly. The practical implication for cache system designs is that the shared bus must form the mechanism for serialization. Where two or more client requests address one line of the shared memory address space, the bus arbitration system determines the order of coherence transitions resulting from the requests. Where a client request is delayed pending allocation of the bus, a cache must grant its snoop exclusive access to its state, forcing the client to wait, in order to avoid deadlock. This mutual exclusion between the client processor and the snoop within the cache (often called *interference*) is a potential source of performance degradation.

Given the assumption of atomicity and the resulting serialization, formal verification of a protocol is based on enumeration of a set of permissible global states, in which coherence is maintained. These are the configurations for each protocol, mentioned in previous sections. Other global states are deemed erroneous, in that they do not guarantee coherence. For example, a global state in which two or more caches have the owned attribute set must be deemed erroneous. The verification process starts from an initial global state (invalid in all caches), and proceeds by a search through the global state space from reachable global states. If a global state is reached which is not a permissible state, the protocol is incorrect. If, when the search terminates, all reachable states are permissible, the protocol is partially correct.

Full correctness also requires verification that the data seen by processors is also the most recently written. This can be verified by augmenting the global state with information about contents of a line, as described by Pong and Dubois in [41]. The authors present a notation to formalize analysis of global state changes, and annotate states with information as to whether copies of data in a line are *fresh*, *obsolete*, or *nodata* (not present). They demonstrate correctness of the Write-once, Illinois, Berkeley, Dragon and Firefly protocols, and for each, derive a set of permissible states that correspond to the configurations described in previous sections of this thesis.

2.3 Protocols Used by Current Processors

In this section, the cache coherence protocols used by a number of widely used commercial microprocessors are described. These protocols are variations of the protocols described in the previous section.

2.3.1 Intel Pentium®

The Intel Pentium® family of microprocessors [34] is widely used in personal computers. It includes on-chip instruction and data caches, with facilities for maintaining coherence between caches in a shared memory multiprocessor system. The cache coherence protocol used for the data cache is based on the write-once protocol described in Section 2.2.2, with some minor differences. First, the Pentium protocol is not write-allocate—on a write miss, the data is written through to main memory without fetching the missed line into the cache. As a consequence, the protocol does not require a bus read-invalidate transaction type. Second, the Pentium bus includes a facility that can be used to detect sharing of a line by other caches. This can be used to optimize the protocol by avoiding the shared stated and the write-invalidate on the first update for private lines. Third, the Pentium bus does not include facilities for directly implementing reflection or intervention. Instead, the same effect is achieved by aborting a miss while the owner cache writes the missed line back to shared memory, then retrying the miss. (This is discussed in relation to the Multibus in Section 2.2.2.) The state transition diagram for the Pentium protocol is shown in Figure 2-10.

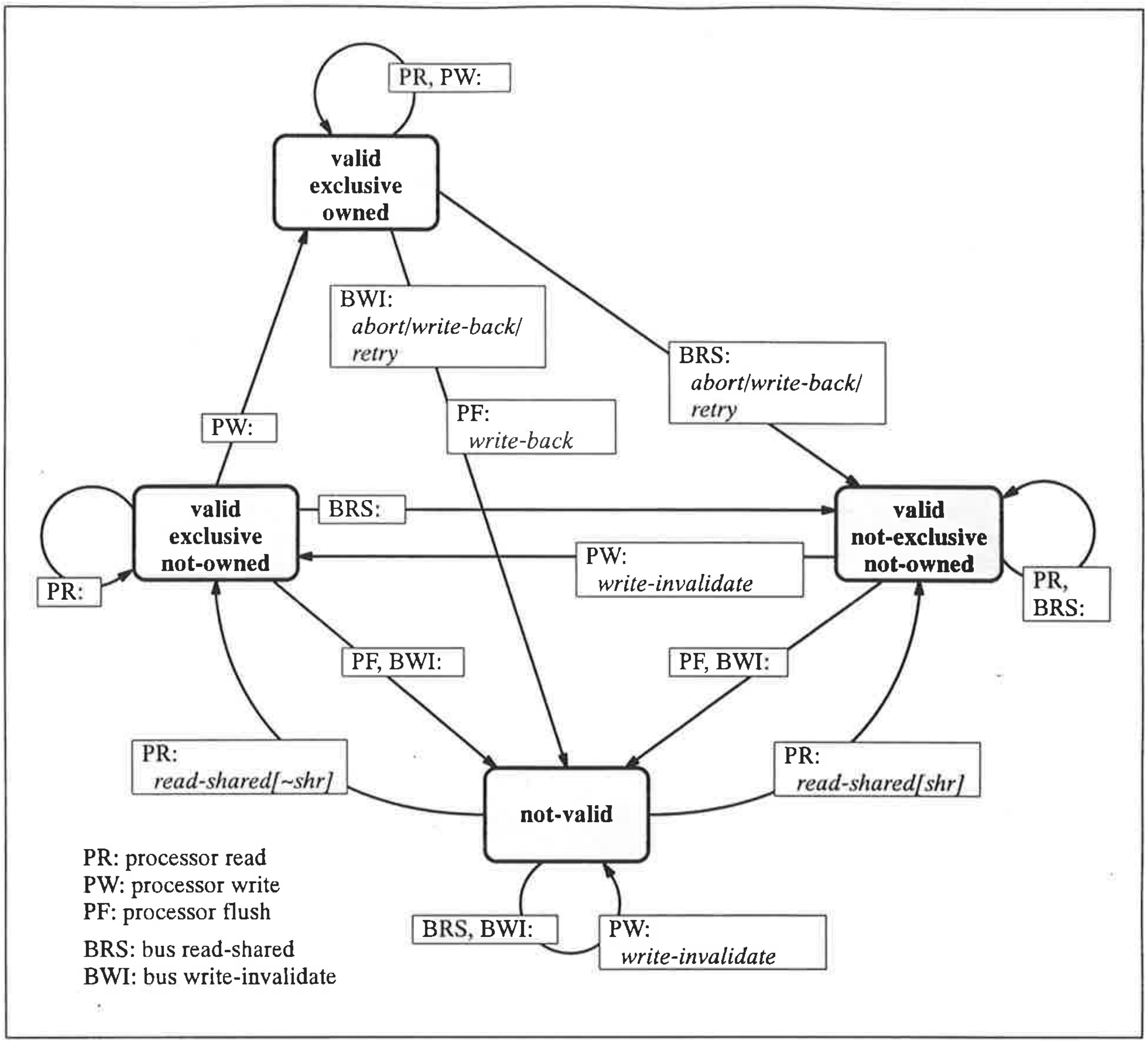


Figure 2-10. The Pentium protocol.

2.3.2 IBM PowerPC

The IBM PowerPC 604 microprocessor [30] implements the PowerPC architecture, developed jointly by IBM and Motorola. The PowerPC 604 uses the Illinois protocol described in Section 2.2.3, with one minor difference. Whereas the Illinois protocol uses intervention and reflection provided directly by the bus protocol, the PowerPC 604 achieves the same effect by aborting a miss while the owner cache writes the missed line back to shared memory, then retrying the miss.

2.3.3 Sun Microsystems UltraSPARC™-II

The UltraSPARC™-II processor [48] is an implementation of Sun Microsystems' SPARC V9 architecture. The UltraSPARC-II includes internal instruction and data caches, and a second level cache implemented with external static RAM, but managed by an on-chip controller. The cache coherence protocol implemented by the controller is based on the MBus protocol described in Section 2.2.6, with two variations. First, the UltraSPARC-II protocol uses a read-invalidate transaction instead of an address-only invalidate transaction on a write hit to a shared line. Second, since the processor provides a block write operation to write a whole line, the UltraSPARC-II protocol includes additional transitions to invalidate a line when a block write is sensed on the bus.

2.3.4 MIPS R4000

The MIPS R4000 family of processors [38] includes the R4000MC, which includes facilities for controlling a coherent second-level cache. The coherence protocol to be used for each page in the address space is determined by configuration bits in the page table entry. A write-invalidate or a write-update protocol can be selected, as well as non-shared and uncached options. The write-invalidate protocol used is the MBus protocol described in Section 2.2.6. The write-update protocol used is similar to the Dragon protocol described in Section 2.2.7, except that sharing detection is not used on update transactions. The modified state transition diagram is shown in Figure 2-11.

2.3.5 DEC Alpha

The DEC Alpha 21164 processor [16] includes controllers for an on-chip second-level cache and an off-chip board-level cache. A five-state write-invalidate cache coherence

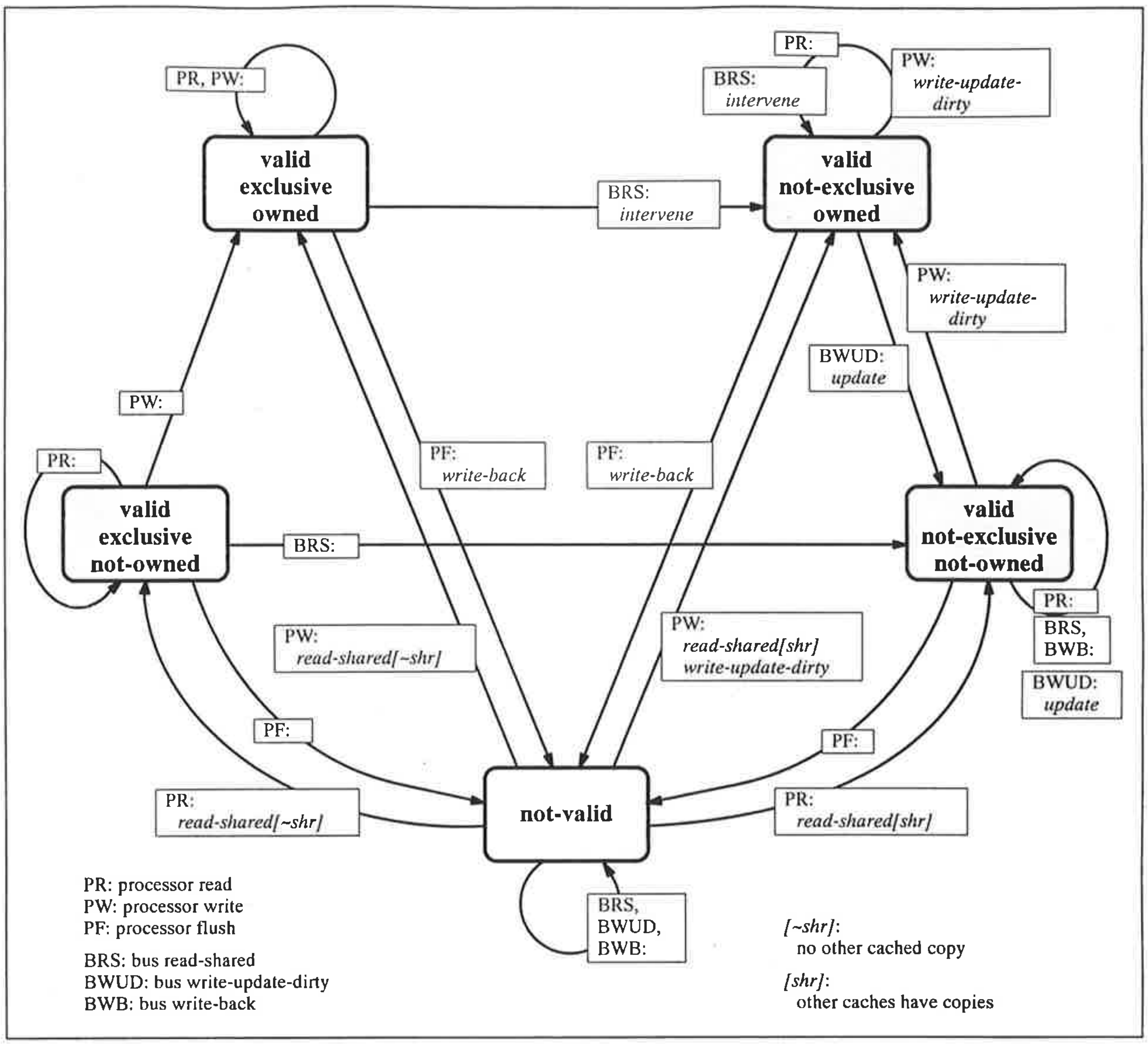


Figure 2-11. The MIPS R4000 write-update protocol.

protocol is followed. The KN470 processor module [24] incorporates an Alpha 21164 processor with a board-level cache, and implements the bus transactions required to support the cache coherency protocol. In order to maintain compatibility with previous processor modules, the KN470 does not implement an address-only invalidation transaction. Instead, write-invalidate transactions are used to inform other caches of modifications to shared lines. Furthermore, the KN470 bus protocol does not provide a mechanism for caches to indicate sharing status during a transaction. While the Alpha

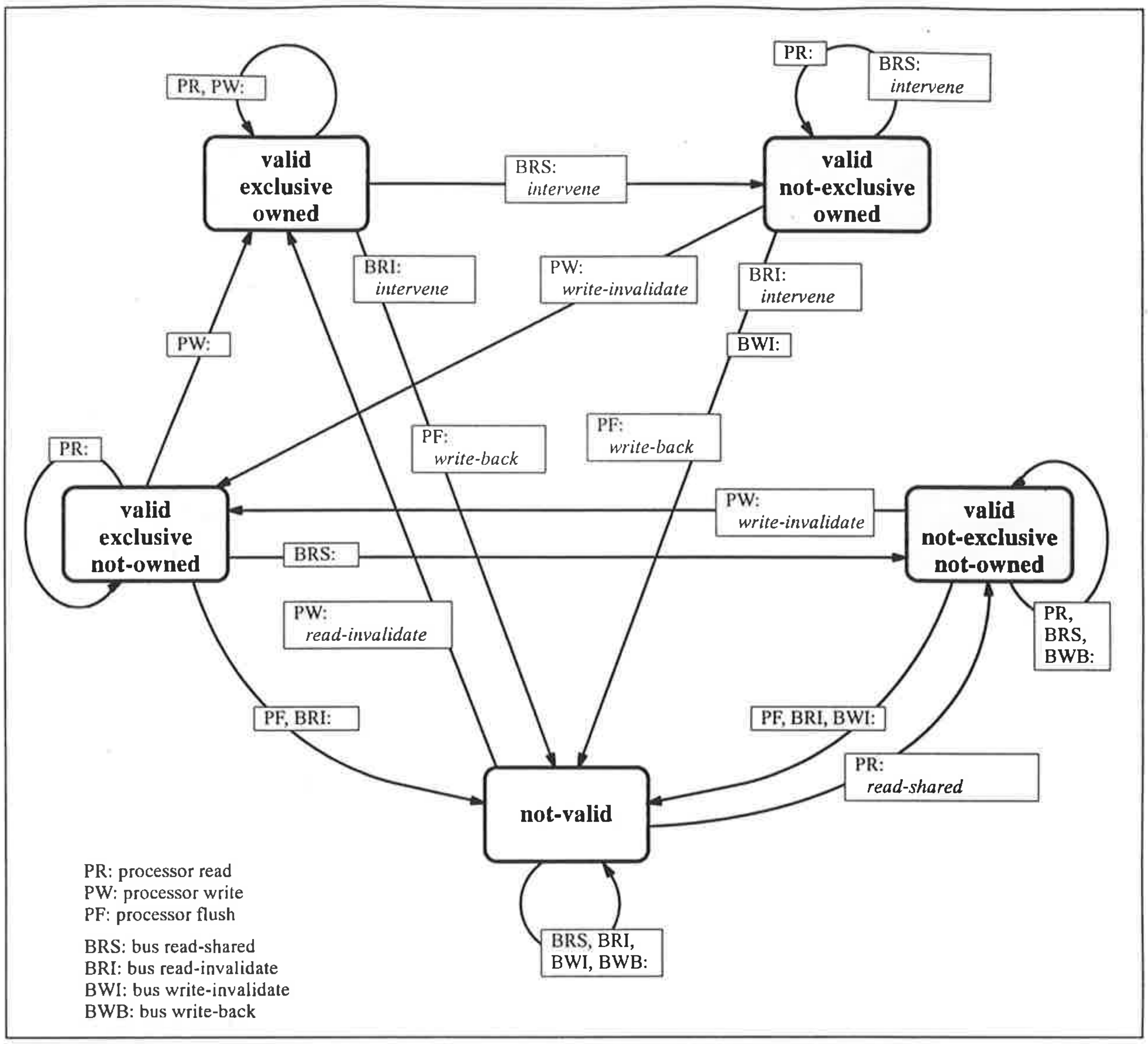


Figure 2-12. The protocol used by the DEC KN470/Alpha 21164 module.

21164 has provision for using sharing status, the lack of bus support prevents use of sharing status in the coherence protocol. The coherence protocol resulting from these limitations is similar to the write-once protocol described in Section 2.2.2, augmented with aspects of the Berkeley ownership protocol described in Section 2.2.5. The state transition diagram is shown in Figure 2-12.

2.4 Proposed Futurebus Cache Coherence Mechanisms

In 1979, the IEEE Computer Society Technical Committee on Microprocessors and Microcomputers set up a working group to develop a new backplane bus protocol. The motivation was that existing bus technology would not be able to meet the bandwidth and functional requirements foreseen at the time. The working group identified support for multiprocessors as a particular deficiency of existing buses, and worked towards a signalling protocol that would support bus-connected shared memory multiprocessors. The first major product of the working group was IEEE Standard 896.1, the Futurebus specification [31], which covered the physical, electrical and basic protocol aspects of the bus. Although it did not specify cache coherence protocols, it did provide hooks on which such protocols could be built. The author was involved as a member of the working group from 1985 until the publication of the standard in 1987.

As part of the 896.1 working group activity, a Cache Coherence Task Group was set up. This later gained full working group status, under the project name P896.2, to develop specifications for higher level protocols for Futurebus, including cache coherence. The author was involved in this group from 1985 until 1989, and contributed an earlier version of the descriptive framework used in Section 2.2. This was subsequently used in the specification of mechanisms to support cache coherence, described in the draft standard produced by the working group [32]. These mechanisms are described here, although not exactly in the form presented in the draft standard, but in a form more consistent with the descriptive framework used in this thesis.

The P896.2 Futurebus support for cache coherence does not actually define a cache coherence protocol. Instead, it specifies a set of mechanisms from which a protocol can be constructed, together with a set of rules governing the use of the mechanisms. The rules allow for alternatives to be chosen, and the particular set of alternatives chosen in an implementation determine the coherence protocol followed. The advantage of this approach is that designers can select protocols suitable for different applications, but still be assured that all modules that conform to the standard will inter-operate. Thus it is possible to construct a heterogeneous system consisting of various forms of copy-back caches, and have them all remain mutually coherent.

This thesis firstly describes the relevant parts of the signalling mechanisms specified to support cache coherence, followed by the rules that govern their use. It then shows how the rules can be applied to implement two copy-back cache coherence protocols, one write-invalidate, and the other write-broadcast. Finally, it discusses the issues of correctness and completeness of the rules.

2.4.1 P896.2 Signalling Mechanisms

Information transfers on Futurebus take place in transactions, consisting of an address transfer, zero or more data transfers, and a disconnection transfer. During an address transfer, a bus master places an address and command on the bus. The commands a master may issue during an address transfer are:

- *Cache-Command*: the master intends to keep a cached copy of the addressed line after successful completion of the transaction.
- *Intent-to-Modify*: the master intends to modify the addressed line.
- *Broadcast*: the transaction will proceed using broadcast signalling, allowing a number of potential slaves to connect and receive data.

Potential slaves examine the address and command, and determine whether to participate in the transaction. They return status information to the master as follows:

- *Cache-Status*: a slave that is neither intervening or reflecting will keep a cached copy of the line after the transaction.

- *Selected*: shared memory recognizes the address and will participate in the transaction. Also asserted by a cache in a broadcast transaction if it will participate.
- *Third-Party*: a cache must participate in the transaction by intervening or reflecting.
- *Intervene*: the third party will participate by intervening, as opposed to reflecting.
- *Busy*: a slave cannot complete the transaction immediately, so the master must abort and retry in a later bus tenure.

If a data transfer follows in the transaction, the master may issue the following command to the participating slaves:

- *Three-Party*: this is a three party transaction. The selected slave should examine the *Intervene* signal, and if it is set, become *disabled* and not participate further. If *Intervene* is clear, the selected slave should become *diverted* and accept data from the reflecting third party cache.
- *Write*: data will be written by the master to the connected slave(s).

At the end of the transaction, the master may issue an additional command:

- *Ownership*: the master will assume ownership, implying that the current owner must relinquish ownership.

These are the basic mechanisms used to build the different kinds of bus transactions referred to in Section 2.2. The P896.2 specification also includes a number of additional commands and status responses, for dealing with error detection and recovery, managing block transfers over line boundaries, and other issues. As this description concentrates on the support for cache coherence, the additional commands and responses are not discussed here, nor are they included in the rules for maintaining coherence. The interested reader is referred to [32] for details.

The way in which the signalling mechanisms are used to implement the transaction types described in Section 2.2 is as follows:

- *read-shared*: during the address transfer the master asserts *Cache-Command* and negates *Intent-to-Modify* and *Broadcast*; during the data transfer the master negates *Write*.
- *read-invalidate*: during the address transfer the master asserts *Cache-Command* and *Intent-to-Modify* and negates *Broadcast*; during the data transfer the master negates *Write*.
- *invalidate*: during the address transfer the master asserts *Cache-Command* and *Intent-to-Modify* and negates *Broadcast*; there is no data transfer.
- *write-invalidate*: during the address transfer the master asserts *Cache-Command* and *Intent-to-Modify* and negates *Broadcast*; during the data transfer the master asserts *Write*.
- *write-update-clean*: during the address transfer the master asserts *Cache-Command*, *Intent-to-Modify* and *Broadcast*; during the data transfer the master asserts *Write*.
- *write-update-dirty*: this can be implemented in exactly the same way as a *write-update-shared* transaction, by ignoring the fact that shared memory is updated as a side-effect.
- *write-back*: during the address transfer the master negates *Cache-Command* and negates *Intent-to-Modify* and *Broadcast*; during the data transfer the master asserts *Write*.

2.4.2 P896.2 Cache Coherence Rules

The rules for maintaining cache coherence specified by P896.2 govern changes in attributes for a line stored in a cache, and changes in connection state of modules during a transaction in which the module recognizes the address. The P896.2 rules use the attributes names *valid*, *exclusive* and *owned*, as in Section 2.2. The P896.2 rules also use the following names for connection states of modules:

- *uns*: unselected (not participating in the transaction)
- *sel*: selected by the address
- *int*: intervening
- *ref*: reflecting
- *dis*: disabled
- *div*: diverted

The rules are formed with a boolean condition on the left hand side, and a resultant assertion on the right hand side. The symbol between them is either "⇒" to denote "shall", mandating change, or "→" to denote "may", allowing change. It is assumed that if a change is neither mandated nor allowed, it is prohibited. The boolean operators used are: "&" for conjunction, "|" for disjunction, and over-bar ("$\overline{\quad}$") for negation. This presentation of the rules follows the same numbering as used in the P896.2 draft specification.

The first set of rules governs line attribute changes made by a cache snoop during a transaction.

(1) Snoop & Cache-Command $\Rightarrow$ $\overline{\text{exclusive}}$

Rule (1) specifies that a cache must relinquish the exclusive attribute if it observes a transaction in which another cache acquires a copy of the line.

(2) Snoop & Ownership $\Rightarrow$ $\overline{\text{owned}}$

Rule (2) specifies that a cache must relinquish ownership if another cache acquires it. This is necessary to ensure that there is only ever one cache that is owner of a line.

(3) Snoop & Intent-to-Modify &
($\overline{\text{Broadcast}}$ & $\overline{\text{owned}}$
| $\overline{\text{Broadcast}}$ & $\overline{\text{Write}}$
| Broadcast & $\overline{\text{sel}}$) $\Rightarrow$ $\overline{\text{valid}}$

Rule (3) specifies that a cache must invalidate a line when a master indicates that it will modify the line, and one of the following:

- the snoop cannot intervene on a non-broadcast transaction (e.g., a non-caching write from an I/O controller), because it is not owner, and thus cannot update its cache with the new data, or
- the transaction is a read-invalidate or an invalidate, or
- the data is broadcast, but the snoop chooses not to update its cache with the new data (see Rules (15) and (16) below).

The second set of rules governs line attribute changes made by a cache when it is master of a transaction initiated in response to a request from its client processor.

(4) Master & Cache-Command $\rightarrow$ valid

Rule (4) specifies that if a cache master indicates it will keep a copy of the line, it may acquire the valid attribute for the line.

(5) Master & Cache-Command & $\overline{\text{Cache-Status}}$ &
(Broadcast
| $\overline{\text{Broadcast}}$ & $\overline{\text{Third-Party}}$
| $\overline{\text{Broadcast}}$ & Intent-to-Modify & $\overline{\text{Write}}$) $\rightarrow$ exclusive

Rule (5) specifies that a cache master may assume the exclusive attribute if it detects that no other cache intends to keep a copy of the line after any of the following transactions:

- a broadcast, or
- a non-broadcast in which there was no third party cache participating, or
- a read-invalidate or invalidate.

(6) Master & Ownership $\Rightarrow$ owned

Rule (6) specifies that a cache master must acquire ownership if it forces another cache to relinquish it. This is necessary to ensure that ownership of a line is not lost, since ownership carries with it the obligation to copy-back a line before replacing it.

The next two rules serve to ensure that line attributes are acquired correctly, allowing only five coherence states for a line.

(7) owned | exclusive $\Rightarrow$ valid

Rule (7) specifies that for a line to become owned or exclusive, it must also become valid.

(8) $\overline{\text{valid}}$ $\Rightarrow$ $\overline{\text{owned}}$ & $\overline{\text{exclusive}}$

Rule (8) specifies that if a line is invalidated, it is no longer owned and no longer exclusive.

The following four rules govern changes that a cache may make to the attributes of a line without incurring bus transactions.

(9) $\overline{\text{owned}}$ $\rightarrow$ $\overline{\text{valid}}$

Rule (9) specifies that a cache may invalidate a line at any time so long as it does not own the line. For example, if a not-owned line is to be replaced or flushed from a cache, it may simply be overwritten. However, if the line is owned, it may not be invalidated without first passing on ownership. (See also Rule (12) below.)

(10) $\overline{\text{owned}} \rightarrow \overline{\text{exclusive}}$

Rule (10) specifies that a cache may relinquish the exclusive attribute for a line it holds at any time so long as it is not owner. In fact, this is overly restrictive. The intention is that a cache may assume a line to be shared without loss of generality. There is no reason why it may not do so when it owns the line, so the following amended Rule (10) is proposed in this thesis, allowing a cache to relinquish the exclusive attribute for a line it holds at any time:

(10a) $\text{valid} \rightarrow \overline{\text{exclusive}}$

(11) $\text{exclusive} \rightarrow \text{owned}$

Rule (11) specifies that a cache that knows a line it holds is not shared by any other cache may assume ownership at any time. Typically it would do this when the client processor issues a write. The fact that the line is not shared means that the cache can proceed with the write locally.

(12) $\overline{\text{Dirty}} \rightarrow \overline{\text{owned}}$

Rule (12) specifies that if the cache can determine that the line is consistent with shared memory, it may relinquish ownership (effectively passing ownership back to shared memory). This may occur in a number of circumstances, such as after a copyback to flush the line, or after a non-broadcast transaction in which the cache reflected.

The next set of rules governs how a cache snoop should set its connection state in response to observed bus transactions, depending on the coherence state of its copy of the line addressed.

(13) $\text{Snoop} \;\&\; \overline{\text{Broadcast}} \;\&\; \text{owner} \Rightarrow \text{int} \;\&\; \overline{\text{ref}} \mid \overline{\text{int}} \;\&\; \text{ref}$

Rule (13) specifies that a snoop that observes a non-broadcast transaction addressing a line which it owns must either intervene or reflect.

(14) $\text{Snoop} \;\&\; \overline{\text{Broadcast}} \;\&\; \overline{\text{owned}} \Rightarrow \text{uns}$

Rule (14) specifies that a snoop that observes a non-broadcast transaction addressing a line which it does not own must remain unselected for the transaction.

(15) Snoop & Broadcast & owner $\Rightarrow$ sel

Rule (15) specifies that a snoop that observes a broadcast transaction addressing a line which is owns must become selected, so that it may participate in the transaction and receive the updated data.

(16) Snoop & Broadcast & valid $\Rightarrow$ sel & $\overline{\text{uns}}$ | $\overline{\text{sel}}$ & uns

Rule (16) specifies that a snoop that observes a broadcast transaction addressing a line of which it has a copy must either become selected to update the copy, or remain unselected. (See also Rule (3) above.)

(17) Snoop & $\overline{\text{valid}}$ $\Rightarrow$ uns

Rule (17) specifies that a snoop that observes a transaction addressing a line of which it does not have a copy must remain unselected.

The following rules govern how the shared memory should set its connection state in response to a transaction that addresses a line it stores.

(18) Shared-Memory $\Rightarrow$ sel

Rule (18) specifies that the shared memory should become selected when it recognizes the address. It will remain selected and transfer data normally, unless its connection state is subsequently changed by application of Rules (19) or (20).

(19) Shared-Memory & sel &
$\overline{\text{Broadcast}}$ & Three-Party & Intervene $\Rightarrow$ dis

Rule (19) specifies that the shared memory must become disabled if a third party cache intervenes on a non-broadcast transaction. The shared memory takes no further part in the transaction.

(20) Shared-Memory & sel &

$\overline{\text{Broadcast}}$ & Three-Party & $\overline{\text{Intervene}}$ $\Rightarrow$ div

Rule (20) specifies that the shared memory must become diverted if a third party cache reflects on a non-broadcast transaction. The shared memory must accept the data transferred on the bus (whether the transaction is a read or a write) and update the stored copy of the line.

The final rule governs the use of the Busy status response from a slave during an address transfer.

(21) Snoop & $\overline{\text{owned}}$ $\Rightarrow$ $\overline{\text{Busy(deadlock-potent)}}$

Rule (21) specifies that a snoop that does not own a line may not return a "deadlock-potent" busy response to the master. This is a busy response reserved for an owner cache that must update shared memory before it can respond to the request from the master. It may occur if a cache designer has chosen to implement the equivalent of reflection by having the cache abort the transaction, update shared memory, and then allow the transaction to be retried. The response is called "deadlock-potent" because of the possibility of deadlock if the cache is also waiting to retry a transaction at some other line in the address space. A protocol must be designed to avoid such deadlock.

2.4.3 Using P896.2 to Implement the Berkeley Protocol

As a demonstration of construction of a cache coherence protocol using the P896.2 rules, a description is presented of the Berkeley protocol, described in Section 2.2.5, specified in terms of the rules. Figure 2-13 shows the state transition diagram for the Berkeley protocol, as presented before, but annotated with the P896.2 rules that govern the transitions. The following commentary describes how each of the rules is applied in the protocol.

(1) Snoop & Cache-Command $\Rightarrow$ $\overline{\text{exclusive}}$

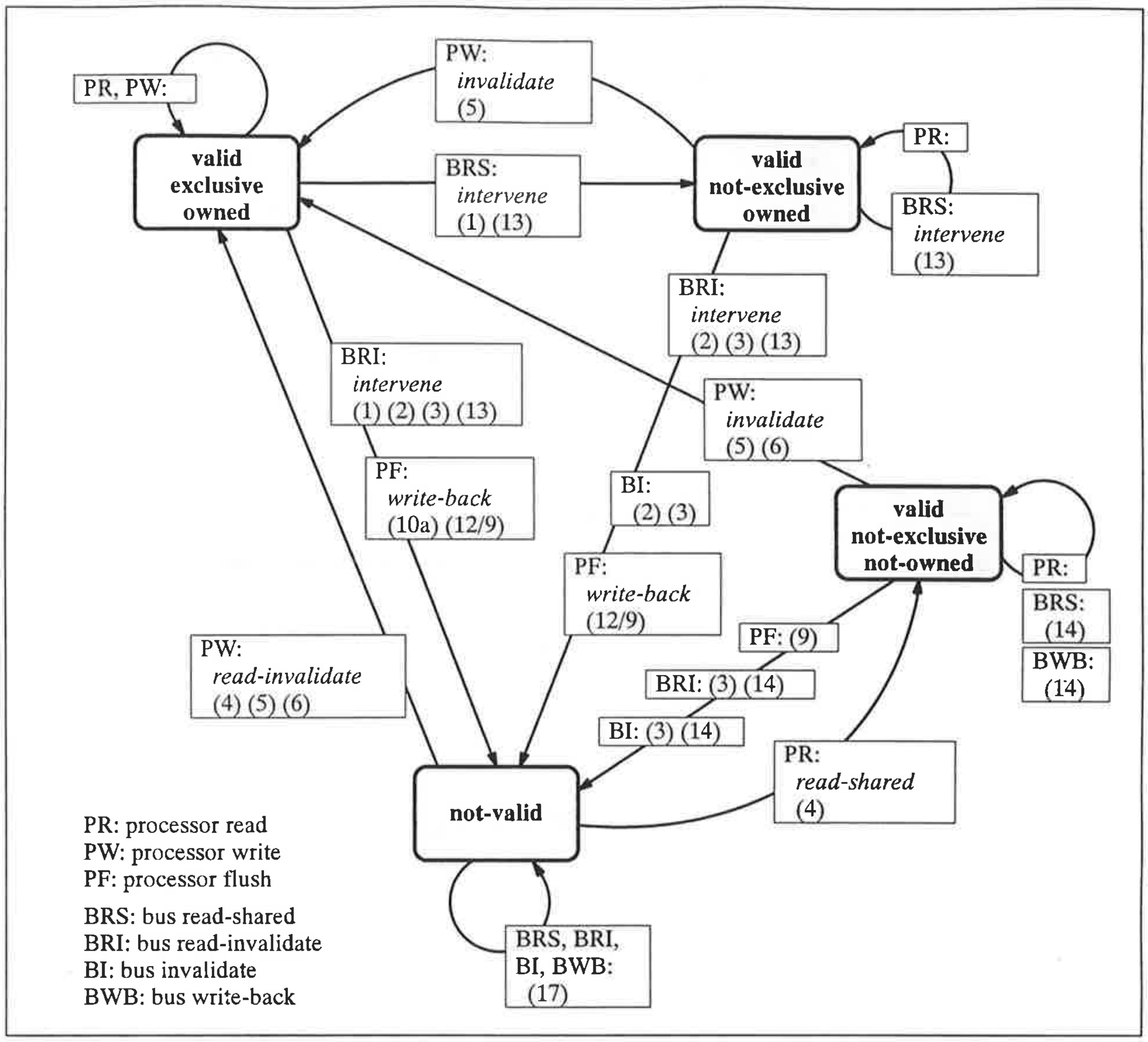


Figure 2-13. The Berkeley protocol transition diagram, annotated with the P896.2 rules that are invoked.

When a snoop observes a read-shared, read-invalidate or invalidate transaction, it relinquishes the exclusive attribute, since the caching master will keep a copy of the line.

(2) Snoop & Ownership $\Rightarrow$ $\overline{\text{owned}}$

When a snoop observes a read-invalidate or invalidate transaction, it relinquishes ownership, passing it on the to caching master.

(3) Snoop & Intent-to-Modify &

$(\ \overline{\text{Broadcast}}\ \&\ \overline{\text{owned}}$

$\mid\ \overline{\text{Broadcast}}\ \&\ \overline{\text{Write}}$

$\mid\ \text{Broadcast}\ \&\ \overline{\text{sel}}\ )\ \Rightarrow\ \overline{\text{valid}}$

When a snoop observes a read-invalidate or invalidate transaction (Intent-to-Modify & $\overline{\text{Broadcast}}$ & $\overline{\text{Write}}$), it invalidates its copy of the line.

(4) Master & Cache-Command $\rightarrow$ valid

When the client processor request results in a miss, the cache fetches the line using a read-shared or read-invalidate, and assumes the valid attribute for the line.

(5) Master & Cache-Command & $\overline{\text{Cache-Status}}$ &

$(\ \text{Broadcast}$

$\mid\ \overline{\text{Broadcast}}\ \&\ \overline{\text{Third-Party}}$

$\mid\ \overline{\text{Broadcast}}\ \&\ \text{Intent-to-Modify}\ \&\ \overline{\text{Write}}\ )\ \rightarrow\ \text{exclusive}$

When a cache issues a read-invalidate or an invalidate transaction (Cache-Command & $\overline{\text{Cache-Status}}$ & $\overline{\text{Broadcast}}$ & Intent-to-Modify & $\overline{\text{Write}}$), it assumes exclusiveness for the line.

(6) Master & Ownership $\Rightarrow$ owned

When a cache issues a read-invalidate or an invalidate transaction, it assumes ownership of the line.

(7) owned $\mid$ exclusive $\Rightarrow$ valid

Covered by Rules (4)–(6).

(8) $\overline{\text{valid}}\ \Rightarrow\ \overline{\text{owned}}\ \&\ \overline{\text{exclusive}}$

Covered by rules (1)–(3).

(9) $\overline{\text{owned}}\ \rightarrow\ \overline{\text{valid}}$

When the client processor causes a not-owned line to be flushed, it is invalidated.

(10a) valid $\rightarrow$ $\overline{\text{exclusive}}$

The exclusive attribute is relinquished when a valid/exclusive/owned line is copied back to shared memory using a write-back transaction.

(11) exclusive $\rightarrow$ owned

Not invoked.

(12) $\overline{\text{Dirty}}$ $\rightarrow$ $\overline{\text{owned}}$

When the client processor causes an owned line to be flushed, it is copied back using a write-back transaction, and is thus no longer dirty. Hence the owned attribute is relinquished.

(13) Snoop & $\overline{\text{Broadcast}}$ & owner $\Rightarrow$ int & $\overline{\text{ref}}$ | $\overline{\text{int}}$ & ref

When a snoop detects a read-shared or read-invalidate transaction addressing a line which it owns, it intervenes.

(14) Snoop & $\overline{\text{Broadcast}}$ & $\overline{\text{owned}}$ $\Rightarrow$ uns

When a snoop detects any transaction addressing a line which it does not own, it remains unselected and does not participate in the transaction.

(15) Snoop & Broadcast & owner $\Rightarrow$ sel

Not invoked.

(16) Snoop & Broadcast & valid $\Rightarrow$ sel & $\overline{\text{uns}}$ | $\overline{\text{sel}}$ & uns

Not invoked.

(17) Snoop & $\overline{\text{valid}}$ $\Rightarrow$ uns

When a snoop detects any transaction addressing a line of which it does not hold a copy, it remains unselected and does not participate in the transaction.

(18) Shared-Memory $\Rightarrow$ sel

The shared memory becomes selected when it recognizes an address.

(19) Shared-Memory & sel &

$\overline{\text{Broadcast}}$ & Three-Party & Intervene $\Rightarrow$ dis

The shared memory becomes disabled on a read-shared or read-invalidate when an owner cache intervenes.

(20) Shared-Memory & sel &

$\overline{\text{Broadcast}}$ & Three-Party & $\overline{\text{Intervene}}$ $\Rightarrow$ div

Not invoked.

(21) Snoop & $\overline{\text{owned}}$ $\Rightarrow$ $\overline{\text{Busy(deadlock-potent)}}$

Not invoked.

Examining the ways the rules are applied, we see that this particular cache coherence protocol arises from the choice of bus transactions constructed from the basic mechanisms, the choice of actions taken by a cache in response to a client processor request, and the choice of whether or not to act on an optional rule (one based on the symbol "$\rightarrow$"). In a homogeneous system constructed with all caches following this protocol, a number of the rules, and factors in some others, are not invoked. However, for a Berkeley protocol cache to be integrated into a system containing caches using other protocols, the cache snoop would have to implement the mandatory parts of all rules that specify a snoop's behaviour. One might debate as to whether this would still be a Berkeley protocol cache. Clearly it would be an extension, with the Berkeley protocol as a proper subset.

2.4.4 Using P896.2 to Implement the Dragon Protocol

As a second demonstration of application of the P896.2 rule, a description is presented of the Dragon protocol, described in Section 2.2.7, specified in terms of the rules. In the Dragon protocol, the write-update-dirty transaction is used to update other caches with a copy of the line, without updating shared memory. The Futurebus mechanisms to not provide a way of ensuring that shared memory does not participate in the broadcast transaction. One alternative when implementing the Dragon protocol is simply to ignore the fact that shared memory copies the broadcast data, as suggested in Section 2.4.1, and implement the protocol exactly as described. The possible penalty here is that participation by shared memory may slow down the broadcast transaction. Given sufficient buffering in the memory system, this can be avoided. Another alternative, applicable in a homogeneous system where all caches use the Dragon protocol, is for the shared memory not to participate in broadcast transactions. This would be an exact implementation of the protocol. A third alternative is to take advantage of participation by shared memory and modify the protocol accordingly.

Figure 2-14 shows the state transition diagram for the Dragon protocol, as presented before, but annotated with the P896.2 rules that govern the transitions. The following commentary describes how each of the rules is applied in the protocol.

(1) $\text{Snoop \& Cache-Command} \Rightarrow \overline{\text{exclusive}}$

When a snoop observes a read-shared transaction, it relinquishes the exclusive attribute, since the caching master will keep a copy of the line. A snoop only observes a write-update-dirty transaction in configurations where it does not have the exclusive attribute, so the rule is not invoked for write-update-dirty transactions.

(2) $\text{Snoop \& Ownership} \Rightarrow \overline{\text{owned}}$

When a snoop observes a write-update-dirty transaction and updates its copy of the line, it relinquishes ownership, passing it on to the caching master.

(3) $$\begin{aligned}&\text{Snoop \& Intent-to-Modify \&}\\ &\quad (\,\overline{\text{Broadcast}} \,\&\, \overline{\text{owned}}\\ &\quad \mid \overline{\text{Broadcast}} \,\&\, \overline{\text{Write}}\\ &\quad \mid \text{Broadcast} \,\&\, \overline{\text{sel}}\,) \Rightarrow \overline{\text{valid}}\end{aligned}$$

Not invoked.

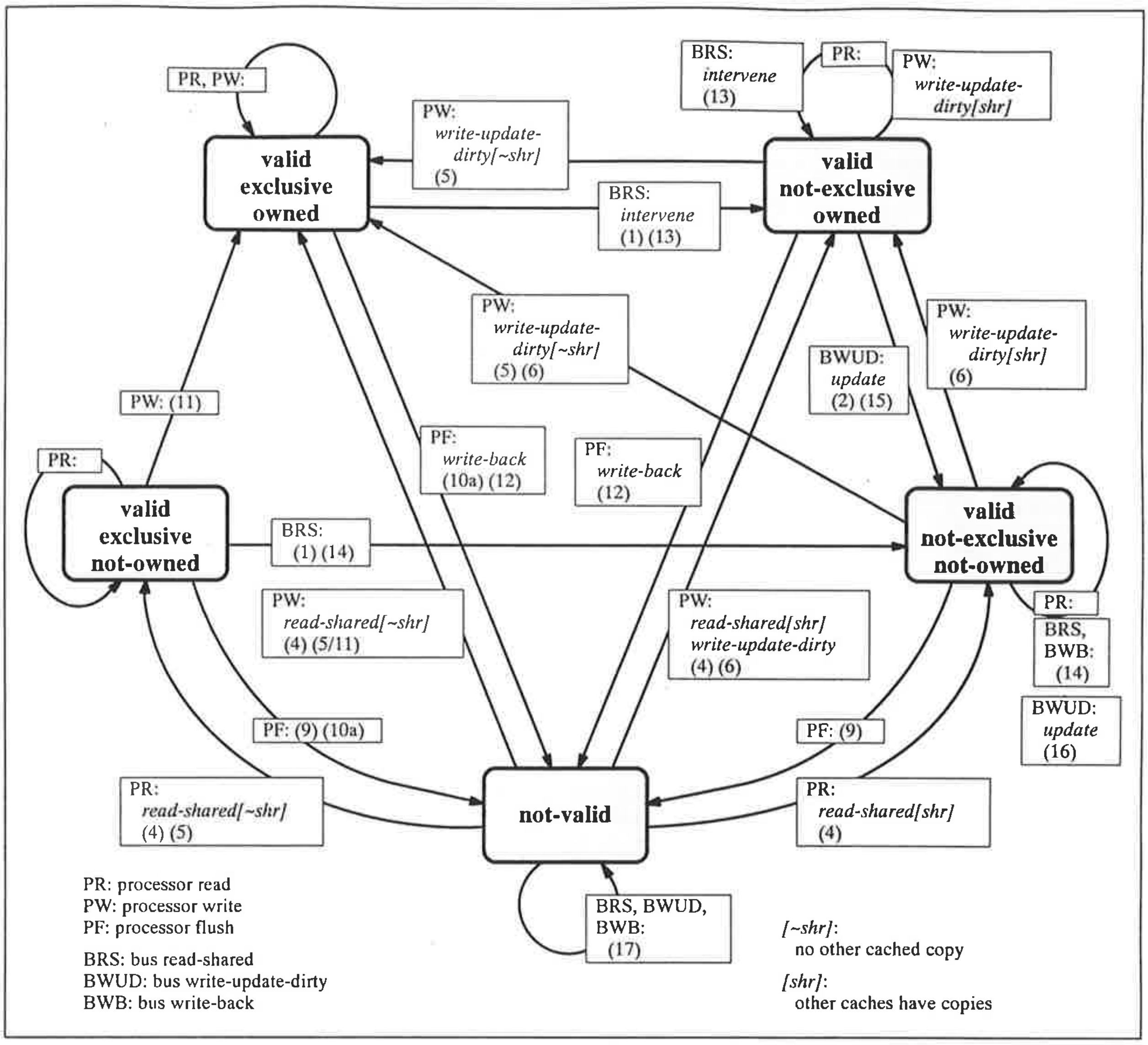

PR, PW:
valid exclusive owned
PW: write-update-dirty[~shr] (5)
BRS: intervene (1) (13)
BRS: intervene (13)
PR:
PW: write-update-dirty[shr]
valid not-exclusive owned
PW: write-update-dirty[~shr] (5) (6)
PW: (11)
PW: write-update-dirty[shr] (6)
BWUD: update (2) (15)
PR:
PF: write-back (10a) (12)
PF: write-back (12)
valid exclusive not-owned
BRS: (1) (14)
valid not-exclusive not-owned
PW: read-shared[~shr] (4) (5/11)
PW: read-shared[shr] write-update-dirty (4) (6)
PR:
BRS, BWB: (14)
BWUD: update (16)
PF: (9) (10a)
PF: (9)
PR: read-shared[~shr] (4) (5)
not-valid
PR: read-shared[shr] (4)
BRS, BWUD, BWB: (17)
PR: processor read
PW: processor write
PF: processor flush
BRS: bus read-shared
BWUD: bus write-update-dirty
BWB: bus write-back
[~shr]: no other cached copy
[shr]: other caches have copies

Figure 2-14. The Dragon protocol transition diagram, annotated with the P896.2 rules that are invoked.

(4) Master & Cache-Command → valid

When the client processor request results in a miss, the cache fetches the line using a read-shared, and assumes the valid attribute for the line.

(5) Master & Cache-Command & $\overline{\text{Cache-Status}}$ &

(Broadcast

| $\overline{\text{Broadcast}}$ & $\overline{\text{Third-Party}}$

| $\overline{\text{Broadcast}}$ & Intent-to-Modify & $\overline{\text{Write}}$) $\rightarrow$ exclusive

When a cache issues a read-shared ($\overline{\text{Broadcast}}$ & $\overline{\text{Third-party}}$) or a write-update-dirty (Broadcast) transaction, and no other cache is sharing the line ($\overline{\text{Cache-Status}}$), the cache assumes exclusiveness for the line.

(6) Master & Ownership $\Rightarrow$ owned

When a cache issues a write-update-dirty transaction, it assumes ownership of the line.

(7) owned | exclusive $\Rightarrow$ valid

Covered by Rules (4)–(6) and (11).

(8) $\overline{\text{valid}}$ $\Rightarrow$ $\overline{\text{owned}}$ & $\overline{\text{exclusive}}$

Covered by rules (1)–(3).

(9) $\overline{\text{owned}}$ $\rightarrow$ $\overline{\text{valid}}$

When the client processor causes a not-owned line to be flushed, it is invalidated.

(10a) valid $\rightarrow$ $\overline{\text{exclusive}}$

The exclusive attribute is relinquished when an exclusive line is flushed.

(11) exclusive $\rightarrow$ owned

The owned attribute is assumed when a client write request hits at an exclusive line and the write is performed locally, or when a client write request misses and the read-shared indicates that no other cache has a copy of the line, allowing the write to be performed locally.

(12) $\overline{\text{Dirty}}$ $\rightarrow$ $\overline{\text{owned}}$

When the client processor causes an owned line to be flushed, it is copied back using a write-back transaction, and is thus no longer dirty. Hence the owned attribute is relinquished.

(13) $\text{Snoop} \mathbin{\&} \overline{\text{Broadcast}} \mathbin{\&} \text{owner} \Rightarrow \text{int} \mathbin{\&} \overline{\text{ref}} \mid \overline{\text{int}} \mathbin{\&} \text{ref}$

When a snoop detects a read-shared transaction addressing a line which it owns, it intervenes.

(14) $\text{Snoop} \mathbin{\&} \overline{\text{Broadcast}} \mathbin{\&} \overline{\text{owned}} \Rightarrow \text{uns}$

When a snoop detects a read-shared or write-back transaction addressing a line which it does not own, it remains unselected and does not participate in the transaction.

(15) $\text{Snoop} \mathbin{\&} \text{Broadcast} \mathbin{\&} \text{owner} \Rightarrow \text{sel}$

When a snoop observes a write-update-dirty transaction addressing a line which it owns, it must become selected and updates its copy.

(16) $\text{Snoop} \mathbin{\&} \text{Broadcast} \mathbin{\&} \text{valid} \Rightarrow \text{sel} \mathbin{\&} \overline{\text{uns}} \mid \overline{\text{sel}} \mathbin{\&} \text{uns}$

When a snoop observes a write-update-dirty transaction addressing a line of which it holds a copy, it becomes selected and updates its copy.

(17) $\text{Snoop} \mathbin{\&} \overline{\text{valid}} \Rightarrow \text{uns}$

When a snoop detects any transaction addressing a line of which it does not hold a copy, it remains unselected and does not participate in the transaction.

(18) $\text{Shared-Memory} \Rightarrow \text{sel}$

The shared memory becomes selected when it recognizes an address. As discussed above, it may or may not actually accept the data transferred on the bus.

(19) $\text{Shared-Memory} \mathbin{\&} \text{sel} \mathbin{\&}$
$\overline{\text{Broadcast}} \mathbin{\&} \text{Three-Party} \mathbin{\&} \text{Intervene} \Rightarrow \text{dis}$

The shared memory becomes disabled on a read-shared transaction when an owner cache intervenes.

(20) Shared-Memory & sel &

$\overline{\text{Broadcast}}$ & Three-Party & $\overline{\text{Intervene}}$ $\Rightarrow$ div

Not invoked.

(21) Snoop & $\overline{\text{owned}}$ $\Rightarrow$ $\overline{\text{Busy(deadlock-potent)}}$

Not invoked.

As with the Berkeley protocol, this cache coherence protocol arises from the choices of bus transactions, cache actions taken in response to client processor requests, and actions on optional rules. What differentiates this protocol from the Berkeley protocol is that a different set of choices is made. The same argument about extending the protocol to integrate with a heterogeneous system also applies.

2.4.5 Summary of P896.2 Options

Since different cache coherence protocols arise from different choices of actions on options in the P896.2 rules, it is useful to summarize the options. This summary is used in development of the Leopard-2 programmable cache controller desicribed in Chapter 4. The following list enumerates the P896.2 rules that allow optional actions and identifies when each option is invoked by the cache coherence protocols.

(4) Master & Cache-Command $\rightarrow$ valid

Although this rule indicates that a caching master is not required to keep a valid copy of a line, in practice there is no need to implement this as an option. A controller design can assume that the line should always be cached and the valid attribute acquired.

(5) Master & Cache-Command & $\overline{\text{Cache-Status}}$ &

(Broadcast

| $\overline{\text{Broadcast}}$ & $\overline{\text{Third-Party}}$

| $\overline{\text{Broadcast}}$ & Intent-to-Modify & $\overline{\text{Write}}$) $\rightarrow$ exclusive

The different protocols vary in when they acquire the exclusive attribute. The Dragon protocols is sensitive to Cache-Status after a broadcast transaction. The Illinois, MBus and Dragon protocols are sensitive to Cache-Status after a read-shared transaction. The original Firefly protocol is sensitive to Cache-Status after both broadcast and read-shared transactions. In all of these cases, the Cache-Status response is used to determine whether the exclusive attribute is acquired. The remaining cases are write-invalidate, read-invalidate and invalidate transactions. All protocols acquire exclusiveness after these transactions, since no other cache asserts Cache-Status in response to them.

(9) $\overline{\text{owned}} \rightarrow \overline{\text{valid}}$

All caches invalidate a not-owned line when it is flushed (eg, during replacement). Thus rule is also used in conjunction with rule (12) when an owned line is flushed. The line is written back to shared memory, allowing the owned attribute to be relinquished according to rule (12), and then the line to be invalidated according to rule (9). The Synapse protocol presents one additional use of this rule, also in conjunction with rule (12): it invalidates a line after a reflection operation causes shared memory to become consistent with the cache.

(10a) valid $\rightarrow \overline{\text{exclusive}}$

None of the protocols make use of this option. The knowledge that a line is exclusive is too good a performance hint to voluntarily relinquish. Hence all of the protocols that use the exclusive attribute only relinquish it when forced to do so according to rule (1).

(11) exclusive $\rightarrow$ owned

All protocols that include the exclusive/not-owned state acquire the owned attribute after a write hit at a line in this state. In combination with rule (5), all protocols except Write-once, Dragon and the original Firefly protocol acquire exclusiveness and hence ownership after a write hit at a not-exclusive line. Similarly, all except Dragon and the original Firefly protocol acquire exclusiveness and hence ownership after a write miss.

(12) $\overline{\text{Dirty}} \rightarrow \overline{\text{owned}}$

This is used by all protocols in combination with rule (9) to invalidate a line after flushing. It is also used by those protocols that implement reflection, to relinquish ownership when shared memory has been updated as a consequence of reflection.

(13) $\text{Snoop} \ \& \ \overline{\text{Broadcast}} \ \& \ \text{owner} \Rightarrow \text{int} \ \& \ \overline{\text{ref}}$
$\quad | \ \overline{\text{int}} \ \& \ \text{ref}$

The choice implied by this rule is whether the owner of a line should intervene or reflect on a read transaction. The Write-once, Illinois and Synapse protocols and the original Firefly protocol use reflection, and the others use intervention.

(16) $\text{Snoop} \ \& \ \text{Broadcast} \ \& \ \text{valid} \Rightarrow \text{sel} \ \& \ \overline{\text{uns}}$
$\quad | \ \overline{\text{sel}} \ \& \ \text{uns}$

The choice implied by this rule is whether a cache should become selected on a broadcast transaction and accept the data, or remain unselected. Only the Dragon and original Firefly protocols involves broadcasts, and in those protocols, a cache with a snoop hit should become selected.

In addition to the rules specified by the P896.2 document, a cache controller must be configured to determine its actions in response to requests from its client processor. All of the protocols described require the same actions in response to the following requests:

- a read hit: perform the read locally with no bus transaction.
- a read miss: perform a bus read-shared transaction to fetch the line and acquire the valid attribute.

- a flush due to replacement: perform a bus write-back transaction if the line is dirty, and invalidate the line.
- a write hit at an exclusive/owned line: perform the write locally with no bus transaction.
- a write hit at an exclusive/not-owned line: perform the write locally and acquire the owned attribute according to rule (11).

The protocols differ in their actions for a write hit at a not-exclusive line and for a write miss. The required bus transactions and attribute changes can be programmed as configuration parameters for the cache controller, as described in Chapter 4.

2.4.6 Correctness and Completeness of the P896.2 Rules

The claim made by the developers of the P896.2 cache coherence specification is that the rules are sufficient to construct a set of interoperable cache coherence protocols. This has not been formally substantiated, however, work by Robinson, with whom the author collaborated in 1988, shows some initial steps towards verification of the rules.

In [42], Robinson describes a transcription of the P896.2 rules into a Prolog knowledge base. These rules are augmented with a further ten rules that formally specify aspects that are informally described in the draft standard. For example, a rule has been added that formalizes the requirement that a snoop assert Cache-Status during a transaction if it intends to keep a copy of the line after the transaction. Robinson also added facts to the knowledge base to represent the state of modules and the bus, and rules to simulate transactions. The stimulus to the simulation was a query that specified a set of client processor requests to be performed. The logic program then inferred the bus transactions required and the coherence state changes necessary to ensure coherence, according to the transcribed P896.2 rules.

In [42], Robinson demonstrates that the logic program correctly simulates the operations (described in [32]) of a heterogeneous system comprising an invalidating copy-back cache, a broadcasting copy-back cache, a write-through cache, a non-caching master and a shared memory. Robinson presents this simulation as the first step towards "find[ing] an invariant expressing cache coherence and proving that any bus transactions complying with the formulæ maintain the invariant." The author of this thesis suggests that one way to formulate such an invariant would be in terms of the global state of the system, as outlined in Section 2.2.11.

An alternate approach to verifying correctness of the protocols also relies on simulation of a system, but is expressed using a behavioural simulation language instead of a logic programming language. The advantage of this approach is that it includes modelling of the sequence of events within a bus transaction, allowing verification of interactions between modules at a more detailed level. This is important for the Futurebus protocol mechanisms, which involve sequences of actions by bus masters, slaves and third parties, each acting in accordance with the P896.2 rules. Not only should the overall effect of the rules be verified, but the cause and effect relationships implied by them must be shown to lead to maintenance of coherence. Conversely, if coherence is not maintained, a verification technique should be able to pin-point the interaction within a transaction that leads to loss of coherence. Compare this with Robinson's approach, in which a transaction is considered as atomic interaction between the participants, with no cause/effect ordering within the transaction being visible. The technique of using detailed simulation to verify correctness of the cache coherence protocols is similar to that used by Wood to test the cache controller circuits designed to implement the Berkeley protocol in the SPUR multiprocessor [55].

The author of this thesis developed a simulation suite to verify the Futurebus protocol mechanisms as part of the collaborative work with Robinson in 1988. The author's work consisted of developing behavioural simulation models, written in the Helix Hardware Description Language [46], of a cache, a Futurebus interface that implemented the basic signalling mechanisms, and a client CPU. This work is described in [3] (reproduced in Appendix C). The cache model was parameterized to allow specification of actions to be taken within the framework of the P896.2 rules. Instances of the modules were interconnected to form a model of a complete bus connected shared memory multiprocessor. The assembled suite was used to simulate operation of caches and Futurebus transactions at the level of individual bus signalling events. The way the suite was to be used to verify the protocols was to stimulate the cache modules with a sequence of a CPU requests and bus transactions that randomly covered the space of possible interactions between modules, and verify the maintenance of coherence as the simulation progressed. Random testing is chosen, as the state space is intractably large. As reported by Wood, a well chosen suite of stimulus vectors gives good coverage and a high degree of confidence in the correctness of the system. Experiments with the suite revealed no cases in which coherence was lost. As well as its use in verifying the correctness of the cache coherence mechanisms, parameterizing of the simulation model made it possible to simulate any of the cache coherence protocols implemented with the basic mechanisms, and to observe the detailed interactions between modules under complex scenarios. This proved very useful as an aid to understanding the dynamics of the cache coherence protocols.

2.5 Performance Evaluation of Coherence Protocols

Having surveyed the published cache coherence protocols for bus connected shared memory multiprocessors, one must ask why there is such a proliferation of protocols. One answer is that they have been developed in response to different perceptions of the optimal points in the space of cost/performance trade-offs. Proponents of the more elaborate protocols argue that the performance gains merit the extra cost, whereas others argue that there are performance penalties associated with the complexity, and that simpler protocols will perform better. This can be most clearly seen in the difference between write-invalidate and write-broadcast protocols. Write-invalidate protocols are simpler to implement, since the data in the cache never needs to be modified as a result of a snooped bus transaction. Only the attributes need to be modified, and then only on the first snoop hit to a line in the cache. On the other hand, a write-broadcast cache needs to provide a data path for broadcast data to be written into the cache data memory. This path needs to be used on every snoop write hit. Proponents of write-broadcast strategies claim that this extra complexity is small, given that a read data path is required to support intervention, and that the performance improvement gained by not invalidating the data in the cache (thus causing future misses when the CPU accesses the data) warrants the extra complexity.

It has also been suggested that write-invalidate protocols result in superior performance over write-broadcast protocols [18]. The premise is that when a cache line is shared, each processor performs several updates before some other processor acquires the line. Under these circumstances, a write-invalidate protocol has the effect of giving a processor an exclusive copy of the line after the first update, allowing it to perform subsequent updates without further bus traffic. A write-broadcast protocol, on the other hand, requires the processor to broadcast each update to all other processors that maintain a cached copy of the line, resulting in increased bus traffic. However, write-invalidate protocols have been shown to have poor behaviour for synchronization operations, such as locks and barriers, which may occur frequently in concurrent programs (see, for example, [26] pp. 699–703). The problem is that the protocol produces a large number of invalidation transactions and consequent read-miss transactions when a lock variable is released. Write-broadcast protocols perform significantly better in this case, requiring many fewer bus transactions. Hence the choice between the two kinds of protocol depends on the frequency of synchronization operations and the patterns of data sharing between processes.

Many of the arguments, such as that of relative implementation complexities mentioned above, rely on qualitative judgements about system and cache design and program behaviour. However, in recent years, computer architects have become increasingly aware of the need for a quantitative basis for their design decisions. Analysis and measurement of system and program behaviour has been shown to lead to more cost effective and higher performance solutions than reliance on conventions, intuitions and qualitative judgements.

These arguments point to the need for quantitative data for use in evaluating performance of cache coherence protocols. With such data, informed design decisions can be made in determining a balance between implementation cost and expected performance for a system. In this section, three approaches for obtaining quantitative data are described: analytical modelling, simulation modelling, and measurement of real systems. Their advantages and disadvantages are discussed.

2.5.1 Analytical Evaluation

The basis behind an analytical performance evaluation technique is a mathematical model of system behaviour, usually involving quantifiable parameters such as probabilities of memory references, and distributions of wait times, access times, etc. The model provides ways of calculating new parameters of interest, such as processor and bus utilization. Ideally, the formulae for calculating such values should be in closed form, allowing a designer to determine quickly values for a variety of alternate design parameters. However, for a mathematical model to be amenable to closed form solution, it must necessarily be relatively simple. Models of computer system behaviour can only achieve such simplicity by being approximate. Hence there is a trade-off between a model's accuracy of prediction and its utility as a design tool. The complexity of more accurate models may require them to be solved using iterative numerical techniques, or may render them totally intractable.

An example of a mathematical modelling technique is presented by Patel in [40]. Patel analyses a system comprising a number of processors each with private cache, and a set of shared memory modules connected to the caches via either a full crossbar or a delta interconnection network. The parameters in the model are:

- N: the number of processors,
- M: the number of memory modules,
- m: the probability that a given cache makes a request to shared memory in a clock cycle,
- w: the average number of cycles for which the cache must wait for arbitration and interconnection contention, and
- t: the time taken for the transaction with shared memory.

Patel derives an expression for U, the processor utilization (the proportion of time for which a processor is not stalled waiting for a cache miss to be serviced), which can be used to compare the relative performance of different cache and system organizations. The analysis relies on the simplifying assumption that cache requests to shared memory are random and uniformly distributed over all memory modules. The author asserts that this is a reasonable characterization of the behaviour of a multiprocessor system, without supplying much supportive evidence or convincing argument.

The formula derived by Patel for U is expressed in terms of the parameter w, which itself is dependent in a non-trivial way on the other parameters. Thus to solve for U, a value for w must also be calculated. Patel cites previous research on the simplest case where $m = 1$ and $t = 1$, stating that no closed form solution had been found, and that solution techniques using Markov analysis required large amounts of calculation. Patel then employs a further approximation in his model, decomposing the wait time w and the transaction time t into $w+t$ independent unit-time requests for service, thus reducing the model to the simplest case described above, and using an approximate analytical solution to this case. He justifies these approximations by comparing the results they yield with results of simulations of a number of system organizations, show-ing that the error is within the confidence bounds of the simulation for parameter values of interest. (Of course, this begs the question of how accurately the simulation mirrors the behaviour of the real system.)

In their description of the Illinois cache coherence protocol [39], Papamarcos and Patel present a performance analysis based on the same technique. Their analytical model is more refined, in that it uses more parameters to describe system behaviour in more detail. A number of these parameters are probabilities of occurrence of events, which must be estimated in order to use the model. There are also other simplifying assumptions, for example, that invalidations only cause one cache to invalidate. The model yields three non-linear equations in three unknowns: the average waiting time per bus request, the real execution time per useful computation time (the inverse of processor utilization), and the average bus utilization. The authors report agreement within 5% between their analytical study of a number of system organizations and trace-driven simulations.

A different approach to analytical modelling of cache coherence protocol performance is presented by Vernon and Holliday in [53], based on use of Generalized Timed Petri Nets (GTPNs), described by the authors in [28]. A GTPN is based on a conventional Petri net, but is augmented with a number of attributes on each transition as follows:

- the probability of firing (which may depend on the marking of the net when the transition is ready to fire),
- the duration of firing (which may also depend on the net marking when the transition fires),
- flags used to determine the probabilities for the next states, and
- resources deemed to be "in use" for the duration of firing (e.g., a bus or memory).

The analytical technique proposed by Vernon and Holliday involves modelling a protocol running under a particular workload with a GTPN. The workload model they use is based on that originally proposed by Dubois and Briggs [17], in which a steam of memory references from a processor is divided into two sub-streams, one for private and shared read-only data, and the other for shared writable data. Vernon and Holliday divide the first of these into the two separate components, private data and shared read-only data. They characterize the workload in terms of relative frequency of accesses from each of the three sub-streams and a geometrically distributed inter-arrival time of requests. The GTPN model driven by this probabilistic workload is further characterized by probability estimates for cache hits, reads (as opposed to writes), replacements, etc, and by the parameters of a hardware system under study, such as the cache hit time and miss penalty.

Once the complete model of protocol and workload has been assembled, the analytical technique treats the GTPN as a stochastic process (by virtue of the probabilistic firing characteristics). Performance measures, based on steady state usage of resources attached to the GTPN, are then determined automatically, derived by finding and analyzing the Markov chain embedded within the GTPN model. The performance measures cited by the authors include bus utilization and speed–up, being the relative execution time of the multiprocessor compared to a monoprocessor with infinite cache.

In [53], Vernon and Holliday present results of analysing a protocol similar to Goodman's write-once protocol, and four variations based on mechanism used in the other protocols discussed in this thesis. They report that, while their analysis indicates improved performance over the basic protocol for each of the variations, the assumed level of sharing and hit rates for private and shared read-only data have significant impact on the derived performance estimates. This indicates that the technique may be suitable for performance studies where the workload characteristics and hardware attributes (apart from coherence protocol) are known in advance, but has serious shortcomings as a more general tool for performance evaluation. Furthermore, the authors comment upon a major disadvantage of the GTPN modelling technique, namely that the state-space size escalates rapidly as model complexity increases. While they suggest investigation into ways of analyzing models with larger state spaces as a future topic for research, they do not appear to have published further in that area.

Vernon, Lazowska and Zahorjan [54] have subsequently proposed an alternative form of analytical model, based on Mean Value Analysis techniques taken from queuing network modelling theory. The approach is "to construct a set of equations that compute the mean values of various performance quantities in terms of the mean values of various model inputs—frequently resorting to iteration when direct calculation is not possible" ([54], p. 310). The model inputs include probabilities of different kinds of access made by the processors in a system, characterizing the workloads run by the processors. From these values, the analytical model provides formulae for calculating secondary probabilities of different actions taken by caches in response to processor access requests, such as satisfying the requests locally, performing broadcast writes or invalidations, etc. The authors then derive a set of formulae that describe the mean total time between memory requests issued by a processor, expressed in terms of the mean processor execution time between requests, τ, and mean response times for different cache, bus and memory actions. The resulting formulae require iterative solution, but converge quickly. The authors compare the results yielded by this mean value analysis with those of the GTPN model, and report close agreement. They also report approximate agreement with some of the simulation studies described in [35] and [2]. However, while they conclude that the mean value model can be customized to other protocols to provide a fast means of predicting approximate performance, they note that there are cases where the technique can produce inaccurate results, and recommend use of more detailed techniques to validate the analyses.

In summary, it can be seen that analytical models of a system, where they exist, can be used as an approximate predictor of system performance. One major difficulty that may confront a system designer is finding a model that takes account of parameters of interest. For example, a designer may wish to explore the performance effects of varying memory reference patterns in the workload. If no available model takes this effect into account, and the designer is not in a position to develop a new analytical model (usually the case), then analytical performance evaluations techniques are not appropriate.

2.5.2 Simulation Based Evaluation

Simulation based techniques for evaluation of performance rely on use of a simulation model of the physical system, that is, a program that attempts to emulate the behaviour of the system over a simulated interval of time. A model for evaluating performance of a cache coherent multiprocessor consists of program modules to emulate the processors, the caches, and the bus and shared memory combination. A processor module generates a stream of memory references that are passed to the corresponding cache module. The cache module processes the memory references according to the cache organization and coherence protocol being simulated, possibly generating requests to be handled by the bus and shared memory module. System performance can be measured with instrumentation embedded in the model and in the run-time system used to execute the model.

One of the advantages of simulation as a means of evaluating system performance is its flexibility. A model can be made as accurate or approximate as required by encoding the appropriate level of detail in the programs that emulate behaviour of the components in the system. However, there is a trade-off, in that more detailed models take correspondingly longer in real time to execute.

Another advantage of using simulation based evaluation over analytical evaluation is that additional system parameters which are difficult to quantify can be included in the model. For example, different workload memory reference patterns can be used, simply by varying the way in which the processor modules in the model generate memory references.

The two main approaches used to generate memory reference streams for simulation models are synthetic workloads, in which a combination of stochastic processes provide successive addresses, and traces of address collected from real programs running on real machines. Synthetic workload generators have the advantage that they provide a simple, compact way of generating a large number of addresses to exercise a simulation model. However, they share the disadvantage with synthetic benchmarks, that they are an approximation to the behaviour of real programs based on intuitions or simplified from measured behaviour. On the other hand, real address traces provide the most accurate characterization of program behaviour, but usually involve storage and management of copious volumes of data, and are difficult to collect. Where useful address traces exist, they are often regarded as proprietary information, jealously guarded by their originators. This is particularly the case for multiprocessor address traces.

An example of use of a simulation model to evaluate performance of cache coherence protocols is described by Archibald and Baer in [2]. Their model consists of a number of processes written in Simula, one for each processor, one for each cache, and one for the system bus, shared memory and cache snoops. The processor modules generate a request stream using a synthetic workload. Memory references are divided into references to private lines and references to sharable lines. For each reference generated by a processor, the choice between these, as well as the choice between a read or write, is made randomly based on parameters input to the model. The cache model handles these requests, again using random choices based on input parameters, simulating the steady state behaviour of the system. Private lines are not explicitly represented, on the assumption that the results of previous research on uniprocessor caches can be applied, and thus probabilistic treatment can be used. Sharable lines on the other hand are explicitly identified, and interactions between caches dealing with shared lines include the line identifier.

Archibald and Baer report measurement of a number of system organizations, comparing the following protocols: write-through, Synapse, Write-once, Illinois, Berkeley, the original Firefly, and Dragon. They identify two main factors which contribute to performance differences between the protocols: the way in which private lines are handled, and the overhead in dealing with shared lines. They conclude that the write-broadcast protocols perform better in handling shared data, but note that there are numerous implementation considerations that might lead a designer to prefer a different protocol.

2.5.3 Evaluation Using Real Systems

One of the most significant disadvantages of both analytical and simulation based techniques for evaluating performance of cache coherence protocols is their failure to accurately characterize the workload driving the caches in a system. Analytical techniques necessarily use very simple models of the workload to make analysis tractable. Simulation based techniques use either synthetic workloads, which again are an approximation to a real workload, or address traces. The problem with address traces, apart from the difficulty in acquiring them, is that they are drawn from execution of a single program, and do not take account of factors such as context switching, operating system execution and I/O activity.

A solution to these problems is to measure the performance of real multiprocessor computer systems using different cache coherence protocols. The difficulty here lies in collecting measurements from sufficiently many systems at different points on the system design space. Most existing designs that differ with respect to coherence protocol also differ in other ways, making it impossible to isolate the effect of coherence protocol upon performance.

In order to address these difficulties, the Leopard-2 Multiprocessor [5] was designed to allow a number of different cache coherence protocols to be implemented, with all other system parameters held constant. The aim was to run a number of workloads that typified different application environments, and to measure performance under each of the cache coherence strategies for each workload. By this means, the effects of operating system and I/O activity would be taken into account. The expectation was that the data gathered from such measurements would be used to either validate or refute previous studies based on analytical and simulation techniques, and to aid designers in choosing a protocol for a new design. The Leopard-2 system is described in detail in the next chapter of this thesis.

2.6 Summary

In this chapter a new descriptive framework for cache coherence protocols is presented, and used as the basis for a survey of previously published protocols. The framework provides a way of making clear the similarities and differences between the protocols. This contrasts with other published surveys which adopt the diverse terminologies of the protocols' original developers, obscuring the similarities and making comparison more difficult.

Secondly, the chapter presents a description of the proposed IEEE P896.2 Futurebus bus protocol mechanisms designed to support implementation of cache coherence protocols in such a way that modules using different protocols can interoperate and maintain coherence. Development of these mechanisms was made tractable by the uniform descriptive framework. This chapter also shows how the published protocols can be implemented in terms of the Futurebus mechanisms, and addresses the issue of verifying correctness of the mechanisms through the vehicle of behavioural simulation.

Finally, noting the importance of quantitatively analysing the performance of systems using different protocols, this chapter examines three approaches to quantitative evaluation, namely analytical, simulation based, and by measurement of real hardware. It shows that, while the first two have a useful role to play in performance evaluation, there is a strong need for application of the third approach, both for the more detailed and accurate data that it produces, and to validate analytical and simulation models.

Chapter 3
The Leopard Multiprocessor

3.1 Background

The Leopard Multiprocessor Project was set up in 1984 to investigate a multiprocessor architecture suitable for use in a networked workstation environment. A series of prototypes based on the bus-connected shared memory architecture was designed and constructed over the period from 1984 to 1992. The first of these, known unofficially as the Leopard-0, was designed in collaboration with an industry partner. It served as a means of gaining experience in designing high performance computer systems, and as the basis of a commercial image display workstation (the QDS-1000 [36]).

The second prototype, the Leopard-1, was a small multiprocessor which included three CPU boards, connected with other boards via the L-bus [4]. This system was designed to test a number of concepts in bus design, in particular, mechanisms to support cache coherence protocols. The author's work in this area lead to involvement with the IEEE P896 Futurebus Working Group. As a result, some of the ideas from the proposed Futurebus were used in the L-Bus, and a number of concepts from L-Bus were adopted in the Futurebus draft specifications, particularly those relating to support for cache coherence. The Leopard-1 was also used as a hardware platform for testing multiprocessor operating systems kernels, to investigate some of the concurrency issues that arise.

The third prototype designed and constructed as part of the Leopard Project was the Leopard-2. It was designed to serve as a platform for investigations into cache coherence protocols, concurrent operating systems and concurrent applications. The previous chapter identified the need for experimental work comparing the performance of the different cache coherence protocols running real applications on real hardware. The Leopard-2 is design to support such experiments, by virtue of having programmable caches attached to the CPUs and connection of a bus monitor to the system bus. The programmable caches allow different cache coherence protocols to be implemented within a uniform system environment (cache size, line size, operating system, etc.). The bus monitor, in conjunction with a logic state analyzer and data acquisition system, allows bus transactions to be traced to observe the behaviour of the cache coherence protocol in detail.

This chapter describes the Leopard architectural framework, and reviews the organization of the Leopard-1 and Leopard-2 systems. It then describes the cache design of the Leopard-2 in detail.

3.2 Leopard Architectural Framework

The three Leopard systems are all based on the Leopard bus connected shared memory architectural framework, illustrated in Figure 3-1. It consists of a pool of homogeneous General Data Processors connected via a broadcast system bus to a Shared Memory system. This basic structure is augmented with facilities for specialized data processing and for input/output.

The General Data Processors in a Leopard system provide a pool of processing resources for the execution of system and application tasks. A task ready to run may be allocated to any General Data Processor, depending on the task scheduling algorithm used by an operating system or some dedicated application. All of the General Data Processors in a particular implementation need not be implemented identically. However, they must provide a uniform execution environment for tasks, so that any task

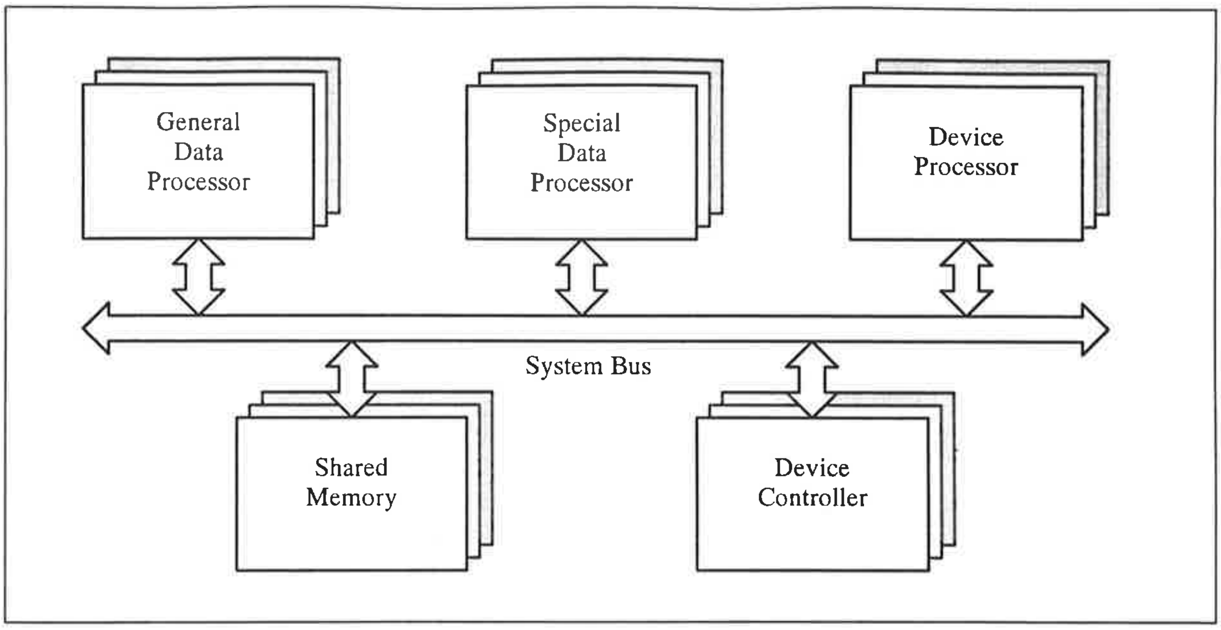


Figure 3-1. Leopard multiprocessor architecture framework.

can run on any processor. This is achieved in Leopard systems by using National Semiconductor NS32000 Series processor components, all of which have the same instruction set architecture [29].

The Shared Memory system provides the primary storage resource, and is shared amongst all processor components. It is used for storage of task code and data, and by virtue of being shared, can be used to implement inter-task communications mechanisms.

The System Bus is a broadcast backplane bus, used to connect all modules in a Leopard system. The architectural framework does not specify a particular bus design, other than requiring it to have high bandwidth, and to support multiple masters using a fair allocation scheme.

A Special Data Processor is an optional component used to optimize performance of some particular processing service. For example, array processors, signal processors, or graphics transformation processors may be included as Special Data Processors.

A Device Processor is a processor to which is attached special hardware for interfacing to external devices, such as file storage media, network connections, graphics displays and other user interfaces. The Device Processor acts as a resource manager, providing device access services to application tasks running on General Data Processors. Access to the services provided is gained using the same inter-task communications mechanisms as are used between application tasks.

A Device Controller is an interface to external devices without an attached processor to act as resource manager. The services of the Device Controller are accessed using conventional Control/Status Registers. Any resource management must be done using tasks running on General Data Processors.

The Leopard architectural framework can serve as the basis for both workstation and network server systems. A workstation can be constructed with a number of processors, a shared memory, a device processor for graphics display and user interface (keyboard, mouse, etc.), and a device processor for network and optional local disk interface. On the other hand, the architecture can be used to create a variety of servers for particular applications. General and special data processors can be used to support both general and application specific computation serving, and various device processors can be used to provide access services for mass storage, network routing, printing, data acquisition, etc.

3.3 The Leopard-1 Multiprocessor

The Leopard-1 Multiprocessor is a small-scale multiprocessor system consisting of three processor boards and a colour frame buffer. The organization of the Leopard-1 is shown in Figure 3-2. Each processor has a block of local memory and basic input/output resources. The frame buffer was designed and constructed by C. Fang (see [19] for a detailed description). A shared memory board was also planned, but proved unneces-

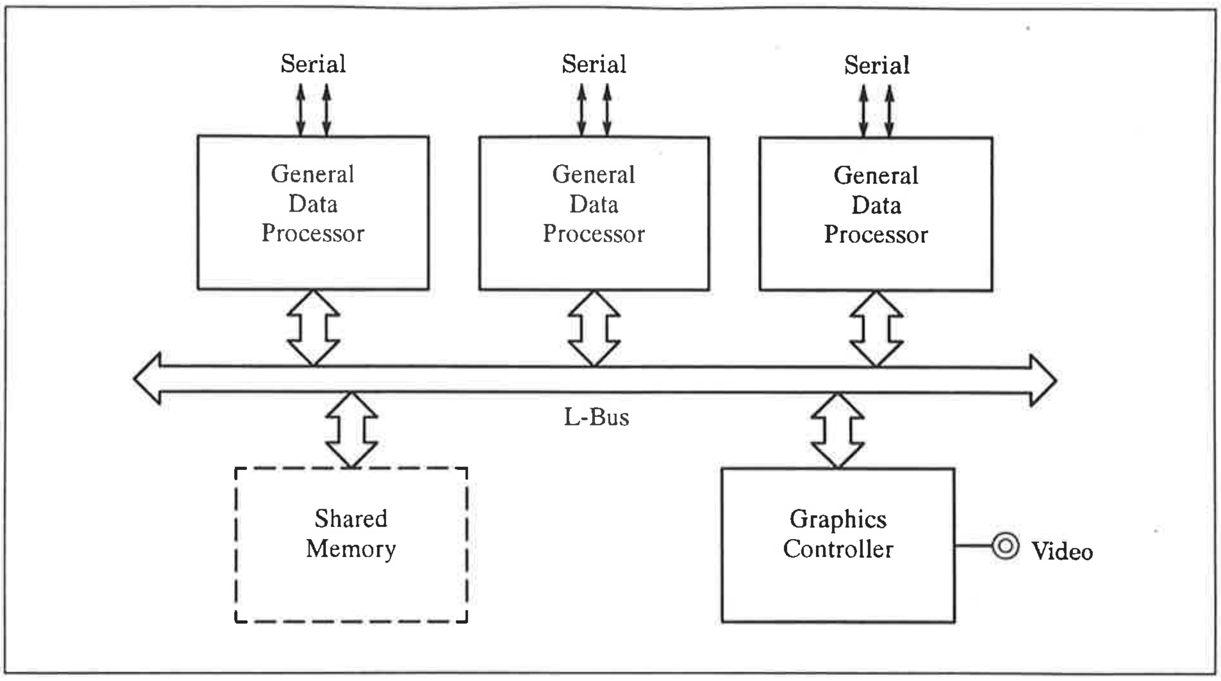


Figure 3-2. Leopard-1 Multiprocessor organization.

sary, as extra memory in the frame buffer proved sufficient to support the experiments for which the Leopard-1 was used.

The L-Bus, used as the system bus in the Leopard-1, was designed to support symmetric multiprocessor operation. Appendix A describes the data transfer protocol in detail. It includes all of the transaction types and mechanisms identified in Chapter 2 to support cache coherence protocols. The bus specification also defines a mechanism for distributing interrupts from device controllers to processors in a Leopard-1 system. The underlying model is that a device controller requests an interrupts at a priority level between 1 and 15 (1 being the lowest priority and 15 being the highest). The processors execute tasks at priority levels between 0 and 15, with 0 being the lowest priority used only for the idle task, and 15 being the highest priority used for an uninterruptible task. When an interrupt is requested at a priority higher than that of the lowest execution priority of any processor, the interrupt mechanism chooses one of the highest priority requests and interrupts one of the lowest priority processors. That processor then reads an interrupt identifier from the interrupting controller, containing informa-tion about its identity and reason for interrupting, and services the request. The inter-rupt protocol is described in detail in [4]. The implementation includes fully distri-buted logic to evaluate the maximum and minimum priorities and to synchronize operations.

The Leopard-1 multiprocessor was constructed in 1986, and used in a number of ex-periments, apart from its use to prove the concepts in the bus design. The Minix operat-ing system [51] was ported to run on a single processor, then subsequently extended to run as a multiprocessor operating system on the complete Leopard-1 multiprocessor. The system also supported experiments in parallel graphics operations.

3.4 The Leopard-2 Multiprocessor

The Leopard-2 Multiprocessor is a multiprocessor workstation based on the general Leopard architectural framework. It includes processor boards each containing an NS32532 processor chip and a large off-chip second-level cache, an I/O device processor for network and mass-storage interface, and an error correcting shared memory. A col-our frame buffer was also designed, but resource limitations prevented its construction. The Chorus multiprocessor operating system [43], designed for this type of architec-ture, has been ported, to serve as the basis for applications software. A bus monitor is also included as part of the hardware, to trace bus transactions as part of the perform-ance measurement experiments.

Figure 3-3 shows the Leopard-2 architecture. Each module is briefly described here, followed by more detailed descriptions of the system bus protocols and General Data Processor design. Detailed descriptions of each of the modules can be found in [5], [6], [7], [8], [9], [10], [11], [12], [13] and [23].

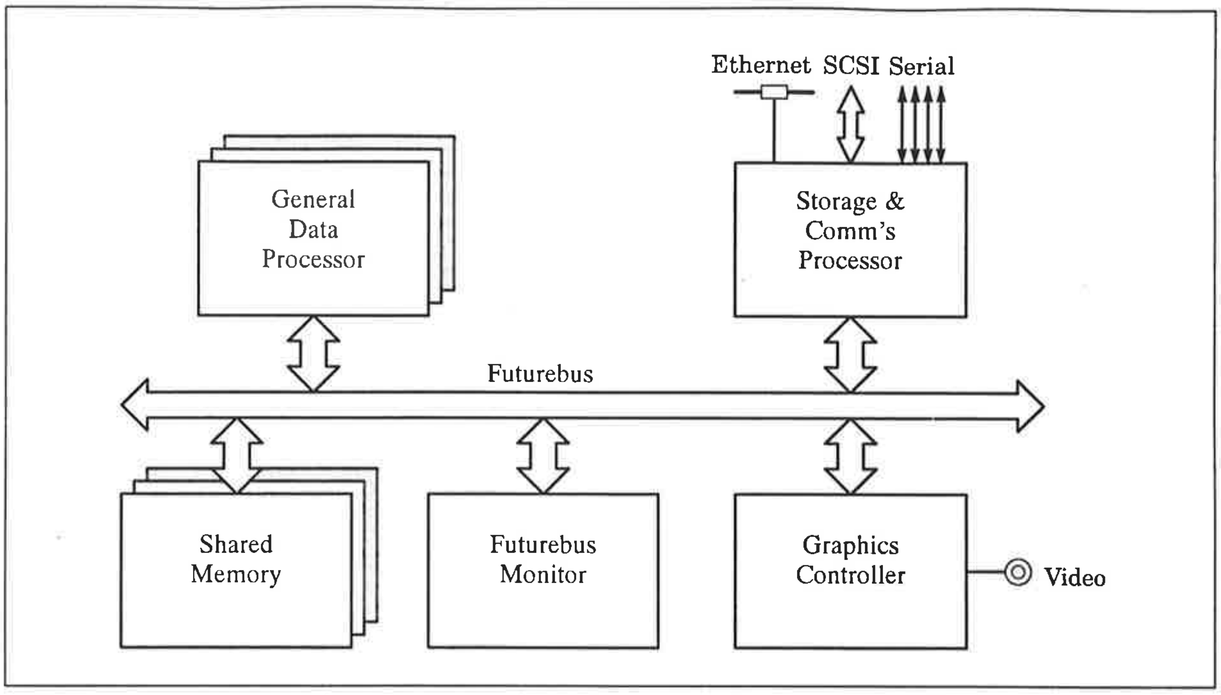


Figure 3-3. Leopard-2 multiprocessor workstation architecture.

The system bus used in the Leopard-2 is based on the IEEE Std. 896.1-1987 Futurebus [31]. At the time of design, this was the only standard bus providing sufficiently high throughput, multiprocessor support and protocols for cache coherence. The ways in which the Leopard-2 bus differ from the full standard are described in Section 3.5.

Each Leopard-2 General Data Processor (GDP) module uses an NS32532 CPU with an NS32381 floating point coprocessor. The CPU includes internal instruction and data caches, and these are augmented with a 512 Kbyte external cache accessed using physical addresses. The external cache incorporates bus snooping hardware to maintain coherence of cached data. The cache controller is designed as a replaceable module, to allow different cache coherence strategies to be applied and evaluated. The GDPs include a local 4 Mbyte error correcting memory for storing processor-local operating system code and data, and two serial ports for diagnostic use. These local resources are not accessible from the system bus.

The Leopard-2 Shared Memory (SM) boards each contain 16 Mbytes of error correcting memory. The memory system uses a three stage pipeline consisting of DRAM access, error checking and bus interface stages. The pipeline is reversible to allow both burst reading and burst writing. By using a 64 bit wide array of static column DRAM devices, the SM can keep up with peak burst transfer rates on the system bus. This is important, as most memory transactions are transfers of cache lines or I/O blocks.

The Leopard-2 Storage and Communications Processor (SCP) is a device processor implemented with an NS32532 CPU. The SCP includes interfaces for Ethernet, SCSI and four general purpose serial ports. It also contains a local 4 Mbyte error correcting memory for local driver code and data, and for I/O buffers. This is supported by high speed block-move hardware for transferring I/O data between buffer memory and shared memory.

The Leopard-2 Graphics Controller was designed as a high resolution colour frame buffer device, providing 2 Mpixels of 8 bits per pixel. The design was adapted from the Leopard-1 graphics controller [19]. The frame buffer is accessible in three areas of the system bus address space, with each area providing a different pixel organization. The first area allows the frame buffer to be used as eight bit-mapped layers, with 32 pixels from a layer accessible in one 32 bit transfer. The second area accesses the frame buffer as an array of 8-bit-deep pixels, with four complete pixels accessible in a 32 bit transfer. The third area allows the use of a collection of raster-op processors to transfer image data between sections of the frame buffer. These flexible access modes allow different types of display operations to be written easily and efficiently.

The Futurebus Monitor is a bus monitoring device used for system debugging and tracing. It contains front panel indicators to reflect the state of bus signals, and connections to a logic analyzer or other tracing instrument for sampling bus states. By programming the instrument with appropriate triggering and sample qualification commands, selective traces of particular bus activity of interest can be captured.

The Leopard-2 is designed to allow comparison of different coherence protocols operating under controlled conditions. The cache data path on each Leopard-2 GDP includes all the resources required for the various protocols, and the controller is a plug-in module. Different controllers can be designed to implement different coherence protocols, and a suite of benchmark programs can be used to measure relative performance. The Futurebus Monitor can be used in combination with a trace capturing instrument to determine the differences in bus behaviour that lead to performance differences.

3.5 The Leopard-2 System Bus

The IEEE Futurebus design is an important part of the Leopard-2 as a platform for experimentation with cache coherence protocols. At the time of design, it was the only bus available which provided a general set of mechanisms allowing all protocols of interest to be implemented. Indeed, as demonstrated in Section 2.4, a number of different protocols can coexist within one Futurebus based system. The Futurebus design makes it feasible to implement plug-in cache controllers for the GDP caches.

Futurebus is an asynchronous 32-bit backplane bus, designed specifically to support high performance multiprocessor systems. The Futurebus standard is composed of two parts. The first part, IEEE Std. 896.1–1987 [31], specifies the electrical signalling levels, the mechanical aspects of board, connector and backplane design, and the basic arbitration, data transfer and system maintenance protocols. The second part, P896.2 [32], was drafted, but never passed as a standard (for numerous reasons, mostly non-technical). It expands on the first part by specifying a CSR architecture, cache coherency protocols, and error recovery mechanisms.

At the time of design of the Leopard-2 system, however, the Futurebus standard was not completely stable, and there were no integrated protocol controller devices available from commercial or other sources. The Futurebus Working Group was considering proposals for extending the standard to include faster arbitration and data transfer protocols. Furthermore, the Leopard-2 does not require all of the facilities specified in the Futurebus standards. For these reasons, the Leopard-2 uses a system bus which is a subset the Futurebus specifications as they stood at the time of design (end of 1988).

In this section, I will outline the differences between the systems bus protocols implemented in the Leopard-2 and those specified in the Futurebus standards.

3.5.1 Arbitration Protocol

The IEEE Futurebus standard specifies a relatively complex but flexible arbitration protocol for allocating bus tenure amongst requesting potential masters. It provides for two classes of modules involved in arbitration: fairness modules and priority modules. The fairness modules in a system are granted access to the bus in such a way that none may be starved by any other. This is achieved by arranging for a fairness module to inhibit further bus requests after a tenure, until all other fairness modules have had pending requests serviced. Priority modules in a system are allocated strictly ordered priority numbers (by the system designer or by configuration software). A priority module is granted bus access in preference to all fairness modules and all other priority modules with lower priority.

The signalling used to implement the arbitration protocol uses distributed asynchronous handshaking, based on the three phase synchronization scheme first proposed for the TriMOSBus [49]. A normal arbitration cycle requires six phases to complete, and an additional three phases when all fairness modules must release their bus request inhibition. The fact that six or nine phases are required is a disadvantage of this scheme. Firstly, on a fully loaded bus, the time taken for an arbitration cycle to complete may be longer than a data transfer transaction. Secondly, when the bus is idle, there is a significant delay before the bus can be allocated to a requester. These two points indicate that a significant proportion of the bus bandwidth may be wasted whilst waiting for arbitration to complete. The IEEE Futurebus Working Group recognized these problems, and considered revising the arbitration protocol to ameliorate them.

In order to avoid the problems, a simplified asynchronous parallel arbitration protocol was developed for use in the Leopard-2. This new protocol is described in detail in Appendix B. The same three phase synchronization mechanism is used, but all arbitration cycles take exactly three phases. Furthermore, none of the operations involved in the protocol are delay operations, as required by the Futurebus protocol to allow distributed priority resolution logic to settle. These changes, in combination, make the Leopard-2 arbitration protocol potentially faster than that specified for the IEEE Futurebus.

3.5.2 Data Transfer Protocol

The Futurebus standard protocol for data transfer includes facilities for use in a diverse range of systems. However, any one system would probably not make use of all of the facilities. This was the case in the Leopard-2, and so a subset of the facilities was used, and only those signals required were actually implemented.

Single address and burst transactions

In a Futurebus system, a transaction may be either single-address or burst-address. A single address transaction consists of an arbitrary number of read and write transfers, in any order, to one single address. This allows implementation of such operations as read-modify-write and write-read-verify as single transactions with only one address transfer. In the Leopard-2, these types of operation are not supported by the NS32532 processors, and the cost of implementing the protocols is avoided. All transactions are treated as burst transfers, with a single quadlet (four-byte word) access treated as a very short burst of one transfer. An atomic read-modify-write operation is implemented as an interlocked sequence of a read transaction followed by a write transaction.

Lane disable signals

The Futurebus specification allows a burst transaction to start at any address and proceed until the master is done or the slave replies with the end-of-data (ED) signal. The specification does not place any restrictions on the lane disable signals during any of the data transfers. So, for example, a master may perform a burst write where only one byte per quadlet is accessed. Having to support such a transaction would greatly increase the complexity of a memory controller, particularly when error checking and correcting circuits are used. The Leopard-2 bus specification states that a transaction may start at any address, with any lanes disabled for the first transfer, but subsequent transfers may not have any lanes disabled.

End-of-data signal and cache line wrap

One of the facilities proposed in P896.2 to support cache systems is "cache line wrap-around." The intention is that a cache may start reading a cache line at the quadlet address requested by its processor, and so satisfy the processor's request immediately. It would then read to the end of the line, and wrap around to the beginning to read the remainder. Two mechanisms are included in the proposal to implement this. Explicit wrap involves the memory slave returning ED at the end of the line, thus forcing the cache master to start another transaction to fetch the remainder of the line. Implicit wrap involves all modules agreeing that when a burst transaction reaches the end of a line, ED is not returned; instead, the burst continues at the beginning of the line. However, the proposal does not specify a way for determining which mechanism is to be used. The Leopard-2 bus solves the problem by specifying that all burst transactions wrap at cache line (16 quadlet) boundaries. Hence the ED signal is not required, and is not implemented. If a master such as an I/O processor needs to transfer a block longer than one cache line, it must break the transfer up into line sized packets. The overhead of doing this is not great, and has the side effect of preventing the bus from being kept by one master for long periods.

Tag bit

The Futurebus specification provides a 32-bit bus for address and data, and also includes an extra tag bit (TG) with parity (TP). The tag signal satisfies the same timing and protocol constraints as the address and data bits, and its use is left unspecified. Since the Leopard-2 has no use for this tag bit, it is not implemented in the Leopard-2 bus.

Parity generation and checking

The 896.1 Futurebus specification includes four byte-parity signals for the address/ data lines and a parity bit for the command lines. It specifies that if parity checking is activated, modules should generate and check for odd parity. If a parity error is detected, an error status should be returned to the bus master. However, the 896.1 document does not specify how parity checking should be activated, nor what error recovery mechanisms apply when a parity error is detected. The P896.2 proposal includes such mechanisms, but they are somewhat complex and expensive to include in a design. For this reason, the Leopard-2 bus specification requires that parity is always generated and checked. When an error is detected, an error status is returned to the master and the transaction is aborted.

Interlocked transaction sequences

Futurebus provides a mechanism to support interlocked sequences of transactions using the LK command bit during the address transfer of a transaction. A bus master may lock any number of slaves by performing successive transactions with the LK bit set. All of the slaves must maintain the lock until a transaction is performed with LK clear, or until the master relinquishes the bus. The scope of a lock on a module (that is, what resources on the module are locked) is module dependent. A difficulty arises with this scheme when a master performs an interlocked sequence of transactions and does not need to perform a subsequent unlocked transaction, and no other module needs to use the bus. In this case, the master remains holder of the bus, and so the interlock is not released. Hence the locked resources remain locked, even though the interlocked sequence is complete. To solve this problem, the Futurebus specification allows the master to perform a special bus arbitration cycle, handing the bus back to itself in order to release the locks. The cost of this is that the master must recognize when such a cycle is required, and then initiate it.

The Leopard-2 specification greatly simplifies the interlock mechanism. As in Futurebus, an interlocked sequence of transactions is signalled using the LK bit during the address transfers. However, in the Leopard-2, the sequence must consist of a read transaction followed by a write transaction at the same address and in the same bus tenure. The interlock lasts for just those two transactions, and the scope of the interlock is the quadlet addressed in the transfers. This scheme greatly simplifies the implementation of interlock management, particularly in copy-back caches, where the lock must extend over a copy the quadlet in a dirty line present in a cache.

3.5.3 System Maintenance

The facilities for system maintenance specified in the Futurebus standard are, like the data transfer protocol, also very general purpose. The Leopard-2 uses a simplified version of these facilities, described in this section.

Memory organization

The Futurebus bus address space of 4 Gbytes is divided into two sections. One section of 32 Mbytes is a structured space reserved for module Control and Status Registers (see below), and the remainder is a linear unstructured space for memory and other resources. Modules may be configured to provide resources at any locations within the memory area. The Leopard architecture adopts this scheme, but in addition reserves the bottom 32 Mbytes of memory address space. Modules must not be configured to provide resources in this area. Instead, modules may use these addresses internally for private resources such as local memory, local I/O devices, etc.

Bus initialization

The IEEE 896.1 specification provides two levels of bus reset, called *bus initialization* and *bus reset*. They are both activated by a pulse on the bus signal, RE. A short pulse signals initialization and a long pulse signals reset. A bus initialization is used to reset the bus interface on each module to recover from bus protocol problems, and does not reset the bus clients. A bus reset is used to recover from serious system-wide problems, and resets the whole system to an initial state. The bus initialize and bus reset protocols both involve alignment processes for the arbitration and data transfer buses. These allow modules to reset all the bus signals to the initial state and to agree when transactions may proceed.

The Leopard-2 system uses a much simplified initialization protocol, providing only a system-wide bus reset. A pulse of at least 10 μs on the bus RE signal is used to activate a system reset. On the leading edge of the pulse, all modules and bus interfaces must return to the initial state and hold that state for as long as RE is asserted. When RE is released, transactions may proceed.

Live insertion and withdrawal

In order to support fault-tolerant and high-availability systems, the Futurebus protocols include an option for inserting and removing modules while a system is active. Live insertion requires that a module be powered and activated by an umbilical cord plugged into the module's front panel before being inserted into the backplane. While being inserted, it must not drive any bus signals. When the module is inserted, the umbilical cable may be removed, and the module must then monitor the bus synchronization lines (strobes and acknowledges) to wait for the beginning of arbitration and data transfer transactions. When a new transaction starts, the newly added module may join in. One of the disadvantages of this scheme is that there is a very short timing window when the new module must assert its synchronization signals on the bus in order to join the transaction reliably. This timing window is specified as an absolute duration (53 ns, based on backplane and transceiver delays), and as such is not "technology independent". The Leopard-2 system has no requirement for live insertion or withdrawal, so the requirements specified by the Futurebus standard to support them are not implemented.

CSR space

The IEEE 896.1 standard and the P896.2 proposal together specify a Control and Status Register (CSR) architecture for Futurebus systems. The architecture is based on *geographic addressing*, where a module's slot number in a backplane determines its CSR addresses in the bus address space. This scheme is extended to allow for addressing modules on multiple buses linked through bus repeaters. Each module is allocated a block of CSR address space, and a number of required registers are specified to provide system maintenance functions.

The CSR architecture used in the Leopard-2 is based on that specified in the IEEE Futurebus standards, but is significantly simplified. Support for multiple buses is not implemented, nor are most of the specified system maintenance registers. Furthermore, addresses in a block transfer in CSR space are not incremented, thus only one register can be addressed per transaction. The interested reader is referred to [6] for further details of the Leopard-2 CSR organization.

Event notification

The IEEE Futurebus standard does not include any backplane signals specifically for handling interrupts. It is expected that I/O interfaces in a high performance system will be managed by an intelligent controller, and there will be no need for time critical interrupts. Instead, the Futurebus provides a mechanism for event notification between modules using the data transfer bus. A module which needs to be notified of events includes 32 Event Registers at a defined location in its CSR space. Any write operation from the bus to one of these registers triggers an interrupt to the module. The specification allows the recording of data with an event notification, and allows queuing of events with the bus busy (BS) signal being used to indicate a full queue.

The Leopard-2 has adopted a simplified version of the Futurebus event notification mechanism, described in detail in [6]. This mechanism is used by device controllers for device interrupts which are not time-critical, and by the operating system to implement message passing and task dispatching.

3.6 The Leopard-2 General Data Processor

The Leopard-2 General Data Processors (L2GDPs) in a Leopard-2 system form the main processing resource for applications and operating system tasks. Each L2GDP includes an NS32532 CPU and NS32381 FPU, which together form the execution unit, a large cache memory, local memory and diagnostic I/O resources, and an interface to the Leopard-2 Futurebus backplane. Figure 3-4 shows the data paths that interconnect these components. A detailed description of the design can be found in [8].

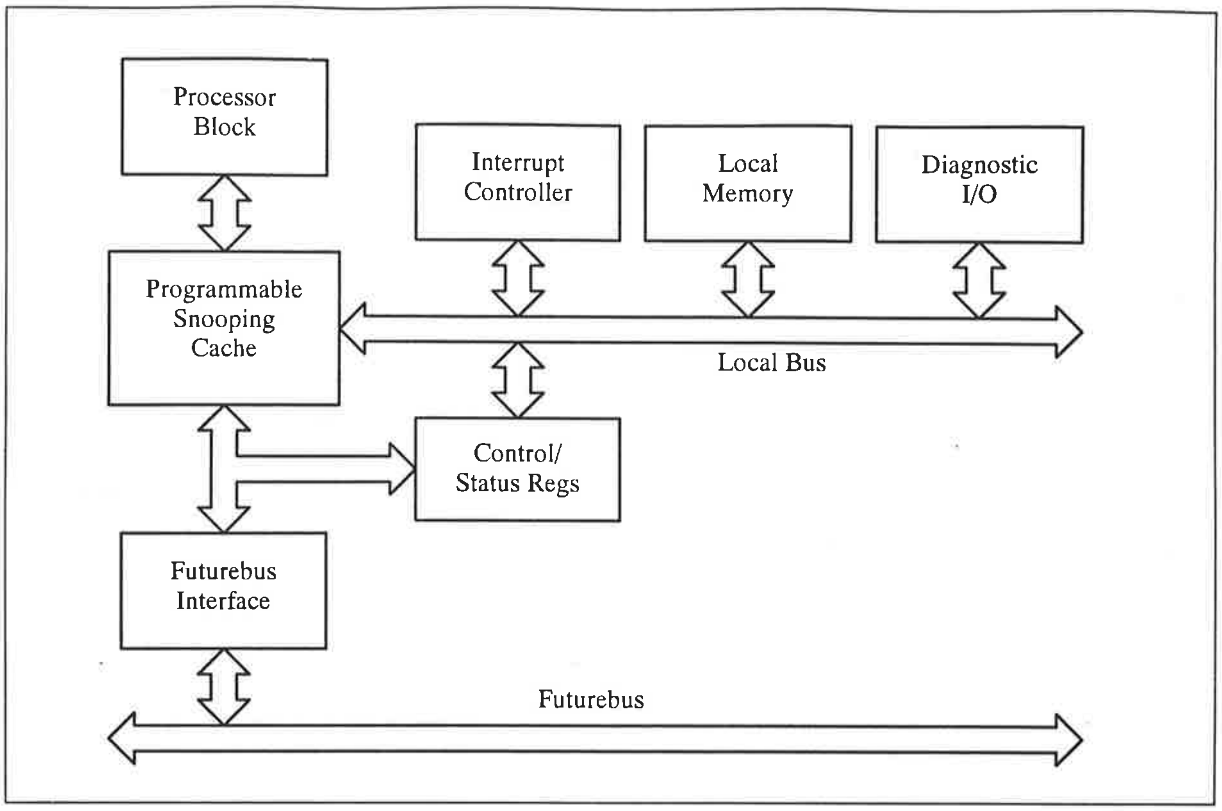


Figure 3-4. The main functional units and data paths of the Leopard-2 General Data Processor (L2GDP).

The processor block contains the execution unit and the clock generator for the L2GDP module. The processor's address, data and control buses connect directly to the cache, allowing high speed transfers of data between the two. The cache is connected to a local bus for access to local resources, and to the Futurebus interface for access to shared system resources.

The local resources consist of an interrupt controller, a local memory block including boot EPROM, static RAM and error correcting dynamic RAM, and a diagnostic I/O interface consisting of two RS-232 serial ports. The local bus is also used for processor access to registers in the L2GDP's CSR block.

The Futurebus interface is used by the cache to provide access to system shared memory and to other modules' CSR spaces. When the cache is not directly controlling the bus as a master, the address and data paths between the cache and the Futurebus interface are used to snoop on bus transactions, in order to implement a cache coherence protocol. This arrangement of buses (a local bus separate from the Futurebus interface data paths) allows internal cache operations and local bus accesses to proceed concurrently with Futurebus snooping.

3.7 The L2GDP Programmable Cache Design

The L2GDP cache consists of separate data path and control sections. The cache data paths contain the memory, buffers and comparators used to process addresses and data. The cache controller is a plug-in module which controls operation of the data path. The reason for making the controller a plug-in unit is to allow different control algorithms to be used, implementing different cache coherence protocols. This is how the Leopard-2 acts as a platform for experimentation with cache coherence protocols.

3.7.1 Cache Organization

The L2GDP cache is a 512 Kbyte 2-way set associative physical address cache. The cache memory can be considered as a two dimensional array of entries, as shown in Figure 3-5. The two columns are called sections, and the 4096 rows are the sets. Each entry in the cache consists of a line of data, a tag indicating the physical address of the line, and a set of attributes.

The NS32532 internal instruction and data caches use a line size of 16 bytes, whereas the P896.2 specification requires that cache line attributes be associated with lines of 64 bytes. For this reason, the Leopard-2 external cache uses a line size of 64 bytes, and divides each line into four sectors of 16 bytes each, corresponding to the CPU internal cache line size. The internal and external caches jointly satisfy the inclusion property that if a 16 byte line is valid in the internal cache, then the 64 byte line of which it is a sector is valid in the external cache.

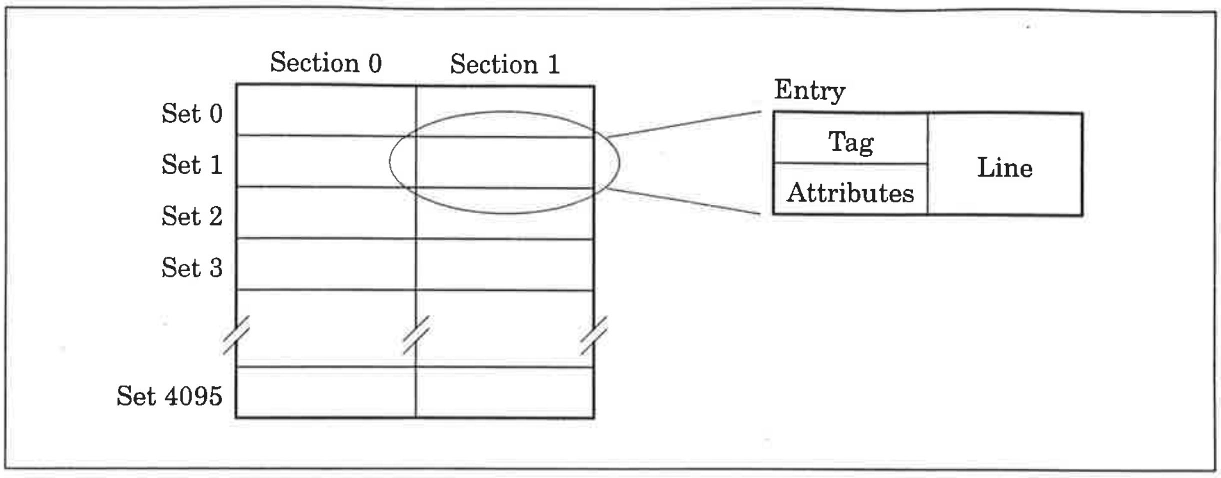


Figure 3-5. Organization of the L2GDP cache memory.

The attributes stored in each entry are shown in Table 3-1. The attributes FB_valid, FB_owned and FB_exclusive are defined by the Futurebus protocol. FB_valid indicates that the line is valid in the cache, and may be read by the CPU. FB_owned indicates that the line has been modified with respect to memory, and the cache must either copy the line back to memory or pass ownership to another cache. FB_exclusive indicates that the line is the only cached copy, and may be written to by the CPU without notifying other caches.

Attribute	Purpose
FB_valid	Line is valid
FB_owned	Line is owned
FB_exclusive	Line is exclusive
Ivalid<3:0>	Sector may be valid in CPU instruction cache
Dvalid<3:0>	Sector may be valid in CPU data cache
Dirty_Sector<3:0>	Sector has been modified
Spare	For use by cache controller

Table 3-1. Leopard-2 cache entry attributes.

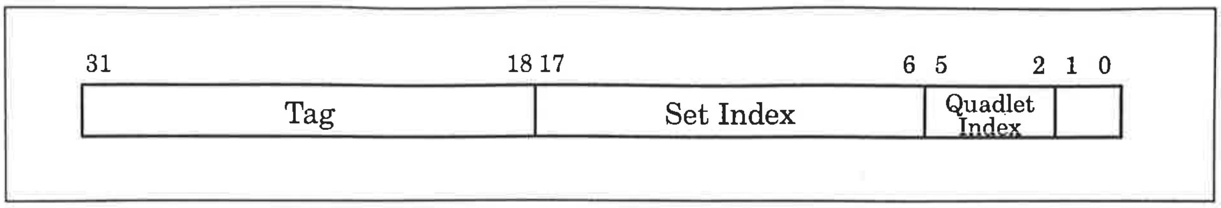


Figure 3-6. Cache address fields.

The attributes Ivalid<3:0> and Dvalid<3:0> are used to maintain the inclusion property. The cache controller can infer from the CPU control signals when a sector is copied into the CPU's internal instruction or data cache, and must set the appropriate attribute bits in the external cache entry. When the external cache entry is subsequently invalidated, the corresponding internal cache entries must also be invalidated using the CPU's invalidation control pins. The Ivalid and Dvalid attributes only indicate that the sector may be valid in an internal cache, not that they definitely are valid. This is because the CPU's invalidation pins can only cause invalidation of a whole set, not an individual entry in the internal cache. Further more, sectors may be invalidated in the internal cache by software CINV (cache invalidate) instructions, without the external cache being notified.

A CPU address is divided into a number of fields for use within the cache, as shown in Figure 3-6. The set index is used to select the set within the cache memory where the addressed line may reside. If the line is stored in the cache, the tag field of the cache entry is set to the tag field of the address. The quadlet index field is then used to select the quadlet within the line.

3.7.2 Cache Data Paths

A block diagram of the L2GDP cache data paths is shown in Figure 3-7. The cache has separate memories for the data, tag and attribute fields of cache entries. Furthermore, the tag and Futurebus coherence attributes of each entry are duplicated for the snoop, allowing the snoop to accesses tags and attributes without interfering with the CPU. How-

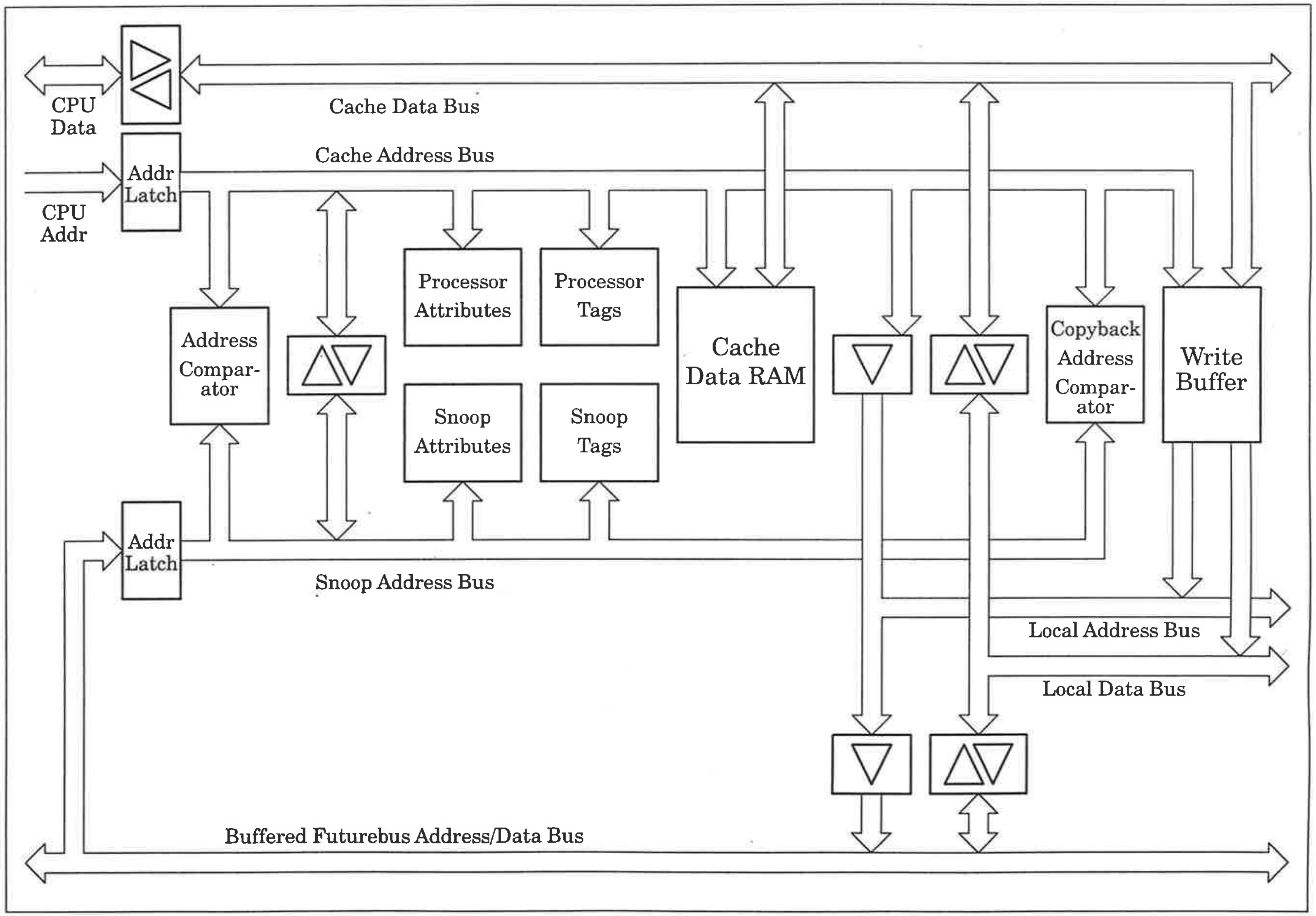


Figure 3-7. Address and data paths within the L2GDP cache.

ever, both copies of a tag or attribute must be updated when a modification is made. The cache also include a write buffer for queuing writes to memory, and for reordering the copy-back and fetching of lines during replacement. When a line is replaced, the controller may copy the line into the buffer whilst waiting for access to the Futurebus. It may then fetch the new line from memory and satisfy the CPU request, before writing the replaced line back to shared memory.

When the CPU initiates a memory reference, the tag and set index fields are latched into the CPU address latch, and the quadlet index bits are input to the cache controller. The latched address may be used as the cache address, along with low order quadlet index bits supplied by the cache controller.

When a Futurebus transaction is started, the tag and set index fields of the Futurebus address are latched in the snoop address latch for use in the snooping function of the cache. These too may be used as the cache address when the snoop address transceiver is enabled in the right direction.

The address comparator allows the cache controller to compare the CPU address with the snoop address. The controller performs this comparison when the CPU requests access to a line while a Futurebus transaction is in progress, or vice versa. If both refer to the same line in the shared memory address space, then mutual exclusion over access to the tags, attributes and data must be enforced, for the reasons discussed in Section 2.2.11.

The data memory is a high speed static RAM array used to store the cached lines. It is organized as two sections, each of 64K 32-bit quadlets. The cache controller selects a section to access, and the quadlet within an entry is selected by the set index and quadlet index fields of the cache address.

The CPU tag memory is an array of high speed tag-RAM devices (a RAM with a built-in comparator). The array is organized as two sections, each with 4096 14-bit tags. The set index field of the cache address is used to select a set of tags. When a cache lookup is being done, the tag-RAM in each section compares the tag field of the cache address with the selected stored tag and generates a hit status output bit. When a line is loaded into the cache, the cache controller selects one section, and the tag field is stored in the selected entry. When a line is being replaced, the cache controller selects a section and disables the tag part of the CPU address latch. The tag-RAM then supplies the tag field to form the address of the line being replaced.

The CPU attribute memory is organized as two sections of 4096 16-bit attribute words. The set index field of the cache address is used to select a set of attributes. The attributes for each section are connected separately to the cache controller. The attribute memory also contains one bit per set for implementing a least recently used (LRU) replacement policy. The cache controller can update this bit on each cache hit.

The snoop tag memory stores duplicates of the CPU tags. It uses the latched snoop address instead of the cache address. Whenever the CPU tag memory is modified, the same modification must be made to the snoop tag memory. This requires disabling the snoop address latch output and enabling the snoop address transceiver to transmit the cache address onto the snoop address bus.

The snoop attribute memory is a partial duplicate of the CPU attributes. Only the FB_valid, FB_owned and FB_exclusive attributes are included, as they are the only ones required by the snoop. This memory must be modified whenever the CPU attributes are modified. Furthermore, if the snoop needs to modify these attributes, the CPU copy must also be modified. This requires disabling the CPU cache address latch output and enabling the snoop address transceiver to transmit the snoop address onto the cache address bus. This address path is also used when the snoop needs to access the cache data memory when it is acting as a third party in a Futurebus transaction.

The write buffer is a 64-entry FIFO for queuing data to be written to memory. The size of the buffer was chosen based on readily available FIFO devices that were sufficient for an entire flushed cache line. Each entry contains a quadlet of data, the address of the data, and a set of flag bits indicating end of a burst, valid bytes within the quadlet, whether the address is a local or Futurebus address, and whether the cache is retaining a copy of the written data. The cache controller generates the flag bits. When the write buffer is used for queuing single-quadlet write-through operations to non-cachable data, the address and data are supplied by the CPU, the end of burst flag is set, and the cache-copy flag is cleared. When the write buffer is used to reorder the copy-back of a line during replacement, the address is derived from the set index and the tag value stored in the CPU tag memory and the data is read from the cache data memory. Only those sectors which are recorded as dirty (having the Dirty_Sector attribute set) need to be written into the buffer. The end of burst flag is set by the cache controller at the end of a run of dirty sectors within the line. The cache-copy flag is cleared

The copy-back address comparator stores the address of an owned cache line when it has been copied into the write buffer for replacement. The cache snoop then uses this comparator to check each Futurebus transaction for a hit with the address of the replaced line. If a hit occurs, the transaction may not proceed and must be retried later. This is an example of the use of the deadlock potent busy response referred to in Rule (21) of the P896.2 coherence rules set, and ensures that coherency is maintained with the line in the write buffer. Because the comparator can only store one address, only one owned line can be queued in the write buffer. However, multiple write-throughs may be queued, and these may coexist with a replaced line in the write buffer.

Note that the lack of snooping on buffered uncachable data does not lead to consistency problems. Suppose a processor A issues a write to an uncachable location, then synchronizes with a processor B, followed by B reading the uncachable location. If the write data is still buffered when the synchronization operation occurs, the buffer in A

is flushed before the synchronization operation completes. The flushing is forced as a result of the read operation that is part of the synchronization operation in A.

The address buffer and data transceiver between the cache buses and the local bus are used for cachable memory accesses to the local memory and for I/O accesses to I/O modules on the local bus. The write buffer outputs also connect to the local bus. The address buffer and data transceiver between the local bus and buffered Futurebus are used for accesses to Futurebus shared memory and Futurebus CSR space. These accesses from the CPU or cache are routed via the local bus. In the case of the snoop participating as a third party in a Futurebus transaction, only the data bus is used.

3.8 Cache Operation

This section firstly outlines the types of requests the CPU on the Leopard-2 General Data Processor makes of the external cache. It then describes the detailed operations that must be performed by the cache data path in response to requests involving cachable data, and to Futurebus transactions observed by the snoop. These operations are managed by the cache controller, which must be implemented as a set of interacting sub-controllers: a CPU request controller, a snoop controller and a write buffer controller. Note that in the actual implementation, many of the operations described here may be performed concurrently. They are presented here sequentially for clarity.

3.8.1 CPU Requirements of the External Cache

The CPU accesses the external cache to fetch instructions, to read and write data, and to fetch and update virtual memory page table entries. From the point of view of the external cache, these can all be treated simply as memory references, without having to distinguish between the different kinds of data. The CPU also accesses non-cachable memory data and I/O device registers using the same memory bus as cachable data, so the external cache must deal with these, transparently accessing the memory or I/O devices as required.

The operations requested by the CPU that must be handled by the external cache are:

- cachable read
- cachable write
- non-cachable read (for non-cachable data or I/O register)
- non-cachable write (for non-cachable data or I/O register)
- interlocked read-modify-write transaction

Reads may be of single words or bursts of multiple words. Writes are always to single words (or parts of single words), since the CPU internal data cache is write-through. The external cache can treat all accesses uniformly as bursts, taking a single word read to be a very short burst of only one word. The NS32532 processor uses bursts to fetch up to a sector of data at a time to fill its internal instruction buffer or an internal cache line. Bursts may start at any address, and in the case of instruction fetches, may only continue as far as the end of the sector (aligned 16 byte block). Bursts for data fetches, however, may wrap back to the beginning of the sector and continue to the word preceding the first fetched in the sector. This allows the CPU to fetch a word immediately to satisfy an internal cache miss, then fill the rest of the internal cache line while the execution unit is processing that word.

The CPU performs an interlocked transaction by first completing any outstanding bus transactions, then asserting the ILO signal. Next, it performs the read followed by the write with no intervening transactions. When they are complete, it negates the ILO signal and resumes normal bus operation. The external cache treats data operated on using interlocked transactions as cachable data, and relies on the cache coherence protocol and the system bus protocols to serialize access. If it has a hit, it treats it as a write hit, acquiring the Futurebus first, then performing the read locally followed by any transaction required for the write. If it has a miss, it acquires the Futurebus, fetches the line as a write miss, then performs the read locally followed by any transaction required for the write.

One problem that arises in a multiprocessor with multi-level caches is that cache coherence must also be maintained between levels of caches. In the L2GDP, the data in the internal cache must be kept consistent with that in the external cache, and hence with shared memory. Since the NS32532 instruction cache is read-only and the data cache is write-through, this reduces to ensuring that if an external cache line is invalidated or updated by the snoop, any portion of that line which is cached internally is also invalidated. The external cache maintains a set of attributes for each line to indicate which of the internal caches may have a copy of each sector. The CPU provides a set of pins to allow the external cache to selectively invalidate internal cache data. It can invalidate the entire instruction cache or data cache, or just a set of either cache. The external attribute bits cannot accurately determine whether a sector is internally cached, for two reasons. Firstly, the data cache is 2-way set associative, but the facility for externally forcing invalidation only allows for an entire set to be invalidated. Thus the external cache attributes for a sector may indicate that it is internally valid, even after it has been invalidated as a side effect of being in the same set as some other sector which was invalidated. Furthermore, the CPU may execute a CINV (cache invalidate) instruction to invalidate an address in an internal cache. The external cache does not include means of detecting this.

3.8.2 CPU Cachable Read and Write Requests

When the CPU initiates a cachable read or write request, it provides the starting address on its address bus. For a read, it expects the data to be provided on its data bus after two clock cycles. For a write, the CPU provides the data at the same time as the address, and expects it to be accepted after two clock cycles. The external cache delays the CPU until it has data available for a read or until it has accepted the data for a write. The operations performed by the external cache are as follows.

Cache lookup

1. The CPU address is saved in the CPU address latch.
2. Arbitration is performed with the snoop controller to gain mutual exclusion to the addressed line of the shared memory address space.
3. The set-index field of the cache address is used to look up the CPU tag and attribute memories in the addressed set. The tag field of the cache address is compared with the fetched tags in each section of the tag memory. Concurrently, the cache address is used to access the line in the data memory, in the optimistic expectation of a hit.
4. If one of the tag memories signals a hit, and the corresponding attribute memory has the FB_valid bit set, a hit in the external cache is indicated, and the corresponding section is selected. The actions for handling a read hit and a write hit are described below.
5. If neither tag memory signals a hit, or if one does but the corresponding attribute memory has the FB_valid bit clear, a miss in the external cache is indicated. If the coherence protocol requires a memory read transaction to fetch the line, a valid line in the cache may need to be replaced. The coherence protocol may also require a memory write transaction. The actions for handling these cases are described below.

Handling a read hit

1. The selected section of the data memory is enabled onto the CPU/cache data bus and the data from the selected entry is accepted by the CPU. The cache

allows the CPU to continue. The CPU provides successive addresses to the cache data memory, accepting each word as it is placed on the CPU data bus by the data memory.

2. When the CPU has accepted the last word in the burst, it terminates the read request.
3. The cache uses bits 4 and 5 of the CPU address to determine the sector number, and sets the Ivalid or Dvalid attribute bit for the line, depending on whether the read request is an instruction or data fetch. The LRU attribute bit is also set or cleared, depending on which section is selected.
4. The updated attributes are written back to the selected section of the CPU attribute memory.
5. The cache releases the mutual exclusion lock on the line. The cache operation is then complete.

Handling a write hit

1. If the coherence protocol requires a bus transaction and the addressed line is in the shared memory address space, the following occurs:
   - 1.1. If the cache has Futurebus tenure, it skips to step 1.6. (The cache has Futurebus tenure if the hit follows immediately from a write miss and tenure was held in order to perform this transaction.)
   - 1.2. Otherwise, the cache must release the mutual exclusion lock on the line requested by the CPU, in order to avoid deadlocking with some other cache which may be waiting to access that line.
   - 1.3. The cache arbitrates for access to the Futurebus.

1.4. The mutual exclusion lock for the line is re-acquired. No arbitration is necessary, as the cache is master of the Futurebus, thus inhibiting any action by the snoop.

1.5. The FB_valid attribute is re-read and checked to see if the cache still has a hit for the line. (The line may have been invalidated by the snoop since the attributes were previously checked.) If the line is invalid, the write hit is turned into a write miss, and operation proceeds as described below (handing a miss). Otherwise operation continues with step 1.6.

1.6. The write buffer is flushed, as described in Section 3.8.5, but without releasing tenure of the Futurebus. (The write buffer must be flushed in order to maintain sequential consistency of memory operations.)

1.7. The address buffers are enabled to pass the address from the cache address bus via the local address bus to the buffered Futurebus address bus, and the data buffers are enabled to pass data from the CPU/cache data bus via the local data bus to the buffered Futurebus data bus.

1.8. The required Futurebus write transaction is initiated, writing from the selected section of the cache data memory. The type of Futurebus transaction used depends on the particular cache coherence protocol being implemented.

1.9. The address and data buffers are disabled and the Futurebus released.

2. If the coherence protocol requires a bus transaction and the addressed line is in the local memory address space, the following occurs:

2.1. The write buffer is flushed, as described in Section 3.8.5. (The write buffer must be flushed in order to maintain sequential consistency of memory operations.)

    2.2. The address buffers are enabled to pass the address from the cache address bus to the local address bus, and the data buffers are enabled to pass data from the CPU/cache data bus to the local data bus.

    2.3. A local bus write transaction is initiated, writing from the selected section of the cache data memory.

    2.4. The address and data buffers are disabled.

3. The cache uses bits 4 and 5 of the CPU address to determine the sector number, and, if necessary, sets the Dirty_Sector attribute bit for the line. The Futurebus attribute values are modified if required by the cache coherence protocol. The LRU attribute bit is also set or cleared, depending on which section is selected.

4. The updated attributes are written back to the selected section of the CPU attribute memory.

5. If the Futurebus attribute values are modified, the new values are written to the snoop attribute memory as follows:

    5.1. Arbitration is performed with the snoop controller to gain access to the snoop attribute memory.

    5.2. The address buffer is enabled to pass the address from the cache address bus to the snoop address bus.

    5.3. The new attribute values are written into the snoop attribute memory.

    5.4. The address buffer between the cache and snoop address buses is disabled, and access to the snoop attribute memory is relinquished.

6. The write enable signal of the selected section of the data memory is enabled, causing it to write the data from the CPU/cache data bus.

7. The cache allows the CPU to continue, and releases the mutual exclusion lock on the line. The cache operation is then complete.

Handling a miss

1. The LRU and FB_valid attribute bits are used to select the least recently used or vacant section of the addressed set.
2. If a read transaction is required and any of the Dirty Sector attribute bits of the selected entry are set, the line in the entry must be replaced. To do this, dirty sectors must be written back to shared memory. The sectors are first copied into the write buffer as follows:
   - 2.1. The cache must release the mutual exclusion lock on the line requested by the CPU, in order to avoid deadlocking with some other cache which may be waiting to access that line.
   - 2.2. If the write buffer is still busy with a previous copy-back or there is insufficient room in the write buffer for the replaced line, the cache must wait until the previous copy-back has been flushed to memory (freeing the copy-back address comparator to hold the address of the newly replaced line) and there is sufficient room.
   - 2.3. The tag field output of the CPU address latch is disabled, and the tag from the selected section of the tag memory is enabled onto the cache address bus in its place. This, together with the set index from the CPU address latch, forms the shared memory address of the line to be replaced.
   - 2.4. The cache must acquire a mutual exclusion lock on the replaced line. This is necessary to avoid a race between the snoop and the cache. Without the lock, if the snoop determines that it must intervene on a

transaction to supply the replaced line, by the time the snoop is granted access to the cache bus the line may no longer be in the cache.

2.5. The FB_valid and Dirty Sector attributes of the replaced line are re-read and checked to see if the line still needs to be copied back. (The attributes may have been changed by the snoop since they were previously checked.) If copy-back is no longer needed, operation continues from step 2.8 below.

2.6. The address of the replaced line is written into the copy-back address comparator.

2.7. Each dirty sector is then read from the selected section of the data memory, word at a time. For each word, a write buffer entry is pushed into the write buffer FIFO. An entry consists of the word of data, its address, four byte-enable bits (all set in this case), a flag to indicate whether the address refers to a local memory or a shared memory location, an end-of-block flag, and a cleared cache-copy flag. The end-of-block flag is set for the last word in a contiguous run of addresses. Adjacent dirty sectors are merged into a block, whereas non-dirty sectors are not written back.

2.8. The mutual exclusion lock on the replaced line is released.

2.9. The tag output from the tag memory is disabled, and the tag field from the CPU address latch is re-enabled, forming the address of the requested line on the cache address bus again.

3. If a read bus transaction is required and the line to be read is a local memory line, it is fetched from local memory as follows:

3.1. The write buffer is flushed, as described in Section 3.8.5. If servicing this cache miss involved buffering a replaced line (described in step 2 above), all write buffer entries up to but not including the replaced line are flushed, otherwise the entire write buffer is flushed. (The write buffer must be flushed in order to maintain sequential consistency of memory operations.)

3.2. The address buffer is enabled to pass the address from the cache address bus to the local address bus, and the data buffer is enabled to pass data from the local data bus to the CPU/cache data bus.

3.3. A local bus burst read transaction is initiated, fetching the line and writing it into the selected section of the cache data memory.

3.4. The address and data buffers are disabled.

3.5. The FB_valid and FB_exclusive attributes of the line are set, and the FB_owned and Dirty Sector attributes are cleared.

3.6. The tag field of the cache address bus is written into the selected section of the cache tag memory.

3.7. Arbitration is performed with the snoop controller to gain access to the snoop tag and attribute memories.

3.8. The address buffer is enabled to pass the address from the cache address bus to the snoop address bus.

3.9. The FB_valid and FB_exclusive attributes of the line are set, and the FB_owned attribute is cleared.

3.10. The tag field of the address is written into the snoop tag memory.

3.11. The address buffer between the cache and snoop address buses is disabled, and access to the snoop tag and attribute memories is relinquished.

3.12. The CPU request is then completed as described above for a hit.

4. If a read bus transaction is required and the line to be read is a shared memory line, it is fetched from shared memory as follows:
   - 4.1. If the cache has Futurebus tenure, it skips to step 4.5. (The cache has Futurebus tenure if the miss follows immediately from a write hit at a shared line, and the line was invalidated by the snoop while the cache was acquiring tenure.)
   - 4.2. Otherwise, the cache must release the mutual exclusion lock on the line requested by the CPU, in order to avoid deadlocking with some other cache which may be waiting to access that line.
   - 4.3. The cache arbitrates for access to the Futurebus.
   - 4.4. The mutual exclusion lock for the line is re-acquired. No arbitration is necessary, as the cache is master of the Futurebus, thus inhibiting any action by the snoop.
   - 4.5. The write buffer is flushed, as described in Section 3.8.5. If servicing this cache miss involved buffering a replaced line (described in step 2 above), all write buffer entries up to but not including the replaced line are flushed, otherwise the entire write buffer is flushed. (The write buffer must be flushed in order to maintain sequential consistency of memory operations.) The Futurebus tenure is not released after flushing the write buffer.
   - 4.6. The address buffers are enabled to pass the address from the cache address bus via the local address bus to the buffered Futurebus address bus, and the data buffers are enabled to pass data from the buffered Futurebus data bus via the local data bus to the CPU/cache data bus.
   - 4.7. A Futurebus read transaction is initiated, fetching the line and writing it into the selected section of the cache data memory. The type of Futu-

rebus transaction used depends on the particular cache coherence protocol being implemented.

4.8. The address and data buffers are disabled.

4.9. If the coherence protocol does not require a write bus transaction to follow the read transaction, or if the write buffer is empty, the Futurebus tenure is released.

4.10. The FB_valid, FB_exclusive and FB_owned attributes of the line are set according to the cache coherence protocol being implemented, and the Dirty Sector attributes are cleared.

4.11. The tag field of the cache address bus is written into the selected section of the cache tag memory.

4.12. The address buffer is enabled to pass the address from the cache address bus to the snoop address bus.

4.13. The FB_valid, FB_exclusive and FB_owned attributes of the line are set in the snoop attribute memory according to the cache coherence protocol being implemented.

4.14. The tag field of the address is written into the snoop tag memory.

4.15. The address buffer between the cache and snoop address buses is disabled.

4.16. The CPU request is then completed as described above for a hit.

3.8.3 CPU Flush

No provision is made in the L2GDP cache for allowing the CPU to specify that a cache line be flushed. Flushing only occurs as part of line replacement on a read or write miss. The reason an explicit flush operation is not provided is that a cache coherence protocol automatically handles the cases where a flush would otherwise be required, such as process migration and I/O operations. In the case of process migration, the process' context is stored in shared physical memory. Reference to it from a new processor involves a cache miss on that processor, and the cache coherence protocol ensures that any modified part of the context is supplied by the old processor using intervention or reflection. In the case of I/O operations, the problem to consider is an I/O buffer written to by a processor and subsequently read by an I/O controller. Again, the cache coherence protocol ensures that, when the I/O controller performs a read transaction to shared memory, the cache supplies any parts of the buffer not yet written back.

3.8.4 Snoop Operation

When a transaction occurs on the Futurebus, the snoop must examine its copy of the cache tags and attributes to determine if it should be involved in the transaction. It must follow the Futurebus cache coherence rules described in Section 2.4.2, customized for whichever cache coherence protocol is being implemented. The operations performed by the snoop are as follows.

Cache lookup

1. The Futurebus address is saved in the snoop address latch.
2. Arbitration is performed with the CPU request controller to gain mutual exclusion to the addressed line of the shared memory address space.
3. A check is made for a write buffer hit, as described below. If a write buffer miss is indicated, operation proceeds.
4. The set-index field of the snoop address is used to look up the snoop tag and attribute memories in the addressed set. The tag field of the snoop address is compared with the fetched tags in each section of the tag memory.

5. If one of the tag memories signals a hit, and the corresponding attribute memory has the FB_valid bit set, a hit in the cache is indicated, and the corresponding section is selected. The actions for handling a hit for different kinds of Futurebus transactions are described below.
6. If neither tag memory signals a hit, or if one does but the corresponding attribute memory has the FB_valid bit clear, a miss in the cache is indicated. In this case, the snoop controller relinquishes mutual exclusion on the addressed line, and remains unselected for the remainder of the transaction.

Hit requiring no action

The snoop relinquishes the mutual exclusion lock on the line and remains unselected for the course of the transaction.

Hit requiring simple attribute modification

If the only action required of the snoop as a result of the transaction is that it relinquish ownership or exclusiveness, it must modify the attributes in both attribute memories at the end of the transaction as follows:

1. The new attributes are written back to the selected section of the snoop attribute memory.
2. Arbitration is performed with the CPU request controller to gain access to the CPU attribute memory.
3. The address buffer is enabled to pass the address from the snoop address bus to the cache address bus.
4. The attributes are read from the selected section of the CPU attribute memory, using the set-index field of the address to access the required set.
5. The attribute values are modified using the new Futurebus attributes, and are written back to the selected section of the CPU attribute memory.

6. The address buffer between the snoop and cache address buses is disabled, access to the CPU attribute memory is relinquished, and the mutual exclusion lock on the line is relinquished.

Hit requiring invalidation

If the action required of the snoop is that it invalidate the line in the cache, then at the end of the transaction it must modify the attributes in both attribute memories and invalidate any sectors of the line cached by the CPU. It does this as follows:

1. The Futurebus attributes are cleared and written back to the selected section of the snoop attribute memory.
2. Arbitration is performed with the CPU request controller to gain access to the CPU attribute memory.
3. The address buffer is enabled to pass the address from the snoop address bus to the cache address bus.
4. The attributes are read from the selected section of the CPU attribute memory, using the set-index field of the address to access the required set.
5. For each sector in the line for which the Ivalid or Dvalid attribute bit is set, a cache invalidation command is applied to the CPU cache invalidation bus.
6. The attribute values are all cleared, and are written back to the selected section of the CPU attribute memory.
7. The address buffer between the snoop and cache address buses is disabled, access to the CPU attribute memory is relinquished, and the mutual exclusion lock on the line is relinquished.

Hit requiring intervention or reflection

If the snoop determines that it must act as a third party in the Futurebus transaction by intervening or reflecting, it does so as follows:

1. Arbitration is performed with the CPU request controller to gain access to the cache address and data buses.
2. The address buffer is enabled to pass the address from the snoop address bus to the cache address bus.
3. The data buffers are enabled to pass data from the cache data bus, via the local data bus, to the buffered Futurebus data bus.
4. For each quadlet of data to be supplied by the cache, the snoop controller supplies the quadlet offset address within the line, and the set-index is supplied from the snoop address latch. This combined address is used to access the data memory, and the data supplied to the Futurebus.
5. At the end of the transaction, the data buffers are disabled.
6. If any Futurebus attributes for the line need to be modified, the snoop performs the appropriate actions as described above.
7. The address buffer between the snoop and cache address buses is disabled, access to the cache buses is relinquished, and the mutual exclusion lock on the line is relinquished.

Hit involving broadcast data

If the Futurebus transaction is a broadcast, and the cache chooses or is compelled to accept the data, it does so as follows:

1. Arbitration is performed with the CPU request controller to gain access to the cache address and data buses.
2. The address buffer is enabled to pass the address from the snoop address bus to the cache address bus.
3. The data buffers are enabled to pass data to the cache data bus, via the local data bus, from the buffered Futurebus data bus.

4. For each quadlet of data broadcast, the snoop controller supplies the quadlet offset address within the line, and the set-index is supplied from the snoop address latch. This combined address is used by the data memory, and the data is supplied from the Futurebus.
5. At the end of the transaction, the data buffers are disabled.
6. If any Futurebus attributes for the line need to be modified, the snoop performs the appropriate actions as described above.
7. The address buffer between the snoop and cache address buses is disabled, access to the cache buses is relinquished, and the mutual exclusion lock on the line is relinquished.

Hit in the write buffer

When a Futurebus transaction occurs, there may be a line in the write buffer waiting to be written back to shared memory. For the line to be in the write buffer, it must have been owned by the cache. Hence, in order for coherence to be maintained, the Futurebus transaction must be aborted and retried after the line is written back. This is achieved by having the cache return a "busy" status during the address transfer part of the transaction. The way in which the snoop checks for and handles a hit in the write buffer is as follows:

1. If there is a replaced line waiting in the write buffer to be copied back to shared memory, the copyback address comparator compares the latched Futurebus address with the address of the replaced line. The Futurebus address stored in the snoop address latch is compared with the address stored in the copy back address comparator.
2. If the addresses refer to different lines in the shared memory address space, a write buffer miss is indicated, and the Futurebus transaction may proceed.

3. If the addresses refer to the same line, a write buffer hit is indicated, and the snoop may not proceed with the Futurebus transaction. Instead, it asserts the BS status signal, causing the transaction master to abort the transaction, relinquish its bus tenure, and retry later. The snoop controller releases the mutual exclusion lock on the line. Since Futurebus arbitration is fair, the write buffer has an opportunity to flush its FIFO, writing the line back to shared memory, before the transaction master retries the transaction.

3.8.5 Asynchronous Writes from the Write Buffer

When the write buffer FIFO is not empty, the write buffer controller asynchronously initiates bus transactions to complete the buffered writes. In addition, it may be forced to flush the write buffer in order to maintain the necessary ordering of memory reads and writes required for sequential consistency. Data buffered in the FIFO may consist of a mix of local memory and shared memory data. There may be a number of non-cachable write-through entries, and at most one write-back entry. The write-back entry is distinguished from the write-through entries by being one or more bursts of data, rather than a single quadlet. The way in which the write buffer controller performs the write operations is as follows .

1. If the entry at the head of the FIFO queue has the flag indicating local memory data set, the write buffer controller performs a local memory write as follows:

    1.1. The write buffer controller arbitrates with the CPU request controller for access to the local bus.

    1.2. If the end of burst flag for the entry is clear, the entry is part of a line being written back to local memory, so the address stored in the copy-back address comparator is cleared.

   1.3. A local bus write transaction is started, with addresses and data being supplied from successive entries from the FIFO.

   1.4. When an entry with the end of burst flag set is written, the local bus transaction is terminated.

   1.5. If the entry now at the head of the FIFO queue is another local memory write, operation repeats from step 1.2.

   1.6. Access to the local bus is relinquished.

2. If the entry at the head of the FIFO queue has the flag indicating local memory data cleared, the write buffer controller writes to shared memory on the Futurebus as follows:

   2.1. The write buffer controller arbitrates with the CPU request controller for access to the local bus, and, if necessary, arbitrates for access to the Futurebus.

   2.2. If the end of burst flag for the entry is clear, the entry is part of a line being written back to shared memory, so the address stored in the copy-back address comparator is cleared.

   2.3. A Futurebus write transaction is started, with the initial address being supplied from the head entry of the FIFO, and data being supplied from the head and successive entries.

   2.4. When an entry with the end of burst flag set is written, the Futurebus transaction is terminated.

   2.5. If the entry now at the head of the FIFO queue is another shared memory write, operation repeats from step 2.2.

   2.6. Access to the local bus and tenure of the Futurebus are relinquished, if not required for subsequent bus transactions.

3.9 Summary

This chapter has described the general architectural framework used for the Leopard Multiprocessors, and presented details of organization and operation of two experimental multiprocessor systems constructed within this framework. A number of significant results flowed from the work on these systems. Firstly, work on the design of the Leopard-1 L-Bus protocols clarified the bus protocol mechanisms needed to support a number of cache coherence protocols with interoperability between them. The work led to contributions to the development of similar mechanisms for the IEEE P896 Futurebus, both in the basic data transfer protocol specified in IEEE Std. 896.1–1987 and in the cache coherence mechanisms defined in the P896.2 draft specification. This in turn led to the development of the Leopard-2 cache design, which can implement a variety of cache coherence protocols. Secondly, the Leopard-2 system described here has been constructed, and is ready for use as a platform for experiments to measure and evaluate cache coherence protocols, and as a shared memory multiprocessor for evaluating concurrent software on such architectures.

Chapter 4
A Programmable Cache Controller for the Leopard-2

4.1 Introduction

The Leopard-2 GDP cache datapath described in Chapter 3 contains the resources required to implement cache coherence protocols such as those described in Chapter 2. One way to vary the coherence protocol implemented by the cache would be to plug in different hardwired cache controller modules, each designed for a specific protocol. However, since the different protocols can be expressed as a set of choices to be made within the options allowed by the P896.2 rules, it is feasible to design a programmable controller that can be reconfigured to implement different protocols.

This chapter outlines the steps needed to design a programmable cache controller that can implement all of the protocols described in Chapter 2, with the exception of the published Firefly protocol described in Section 2.2.9. The published Firefly protocol cannot be directly implemented, as the Futurebus specification fixes the cache line size at 64 bytes. For this reason, and the reasons discussed at the end of Section 2.2.9, the published Firefly protocol is not included in the following discussion. The original Firefly protocol, described in 2.2.8, can be implemented by the reconfigurable cache controller.

The outline of the programmable cache controller is followed by a description of a simulation model of the Leopard-2 system developed using the hardware description language VHDL [14, 33]. The model includes a detailed specification of the behaviour of the programmable cache controller. Through simulation of the model, verification of correct operation is performed.

4.2 Cache Controller Configuration Parameters

The cache controller for the Leopard-2 GDP cache is parameterized to implement each of the cache coherence strategies under consideration. This is done by including a configuration register in the controller, writable by system software. The register contains parameters that indicate which choice should be taken for each of the options described in Section 2.4.5. Table 4-1 shows the register parameters, the corresponding P896.2 rules and the allowed values.

Parameter mnemonic	Values	Parameter meaning	P896.2 rule
excl_depends_on_CS_-on_read_shared	yes/no	If yes, cache checks Cache-Status on bus read-shared to determine whether to acquire exclusive attribute; If no, cache does not acquire exclusive attribute	5
tr_write_hit_shared	invalidate/ read-invalidate/ write-invalidate/ write-update-dirty/ write-update-clean	Transaction performed by cache on write hit at shared line	
owned_on_write_hit_shared	yes/no	If yes, cache acquires owned attribute on write hit at not-exclusive line; If no, cache does not acquire owned attribute	5 & 11
excl_depends_on_CS_-on_write_hit_shared	yes/no	If yes, cache checks Cache-Status on write hit at shared line to determine whether to acquires exclusive attribute; If no, cache always acquires exclusive attribute	5
tr_write_miss	read-invalidate/ read-shared*	Transaction performed by cache on write miss	
reflect_on_read_shared	yes/no	If yes, owner snoop reflects on bus read-shared; If no, owner snoop intervenes	13
inval_if_third_party	yes/no	If yes, snoop invalidates after becoming a third party on bus read-shared; If no, snoop does not invalidate	12 & 9
sel_on_broadcast_hit	yes/no	If yes, snoop becomes selected when it gets a hit on bus broadcast; If no, snoop remains unselected	16

* if Cache-Status asserted during the read-shared transaction, cache follows it with the transaction specified by tr_write_hit_shared.

Table 4-1. Configuration parameters for a reconfigurable cache controller.

Using this parameterization of the cache controller, the coherence protocols are implemented with different sets of parameter values, as shown in Table 4-2. The binary encoded values of these parameters are used in the finite state machines used to implement the cache controller, to determine the actions taken at choice points in the control algorithms described in Section 3.8.

Parameter	Write-once	Illinois	Synapse	Berkeley	MBus	Dragon	Original Firefly
excl_depends_on_CS_-on_read_shared	no	yes	no	no	yes	yes	yes
tr_write_hit_shared	write-inval	inval	read-inval	inval	inval	write-update-dirty	write-up-date-clean
owned_on_write_hit_shared	no	yes	yes	yes	yes	yes	no
excl_depends_on_CS_-on_write_hit_shared	no	no	no	no	no	yes	yes
tr_write_miss	read-inval	read–inval	read-inval	read-inval	read-inval	read-shared	read-shared
reflect_on_read_shared	yes	yes	yes	no	no	no	yes
inval_if_third_party	no	no	yes	no	no	no	no
sel_on_broadcast_hit	no	no	no	no	no	yes	yes

Table 4-2. Configuration parameter values to implement different cache coherence protocols.

A programmable cache controller has a number of important advantages over separate hardwired controllers. The most obvious advantage is the reduced cost and ease of operation, making experiments with different cache coherence protocols simpler and faster to perform. However, the fact that the controller can be reconfigured dynamically is important in a production system running a variety of different workloads. Early evidence for this can be seen in cache controllers for small multiprocessor workstations and servers, which allow a choice of copy-back, write-through or no caching, selectable for each page of a virtual address space. (The cache controller in the Intel Pentium® processor [34] is an example.) The extension to this scheme, provided by a dynamically reconfigurable cache controller as described above, is to select the most appropriate cache coherence protocol for each address space, depending on the memory referencing behaviour of the processes using the address space. A configuration vector may be associated with each address space, and when a process using the address space is scheduled onto a processor, the configuration vector is loaded into the processor's cache controller.

4.3 A VHDL Model of the Programmable Cache Controller

There are a number of important motivations for developing a model of a complex hardware system. Several are discussed by the author in [14] (pp. 2–3), the most relevant being to specify the requirements and functionality of the system completely and unambiguously, and to allow testing and verification of the design through simulation. (Note that the hardware design community uses the term "verification" to include testing. The term does not necessarily imply use of formal mathematical methods to prove that the model meets a specification.) This section presents a model of the Leopard–2 Multiprocessor developed by the author using the standard hardware description language VHDL. The model was developed to specify the behaviour of a programmable cache controller for the Leopard-2 GDP using the cache configuration parameters described in Section 4.2, and to allow simulation experiments to be performed to verify the designs of the Leopard-2 GDP cache and cache controller. The approach taken to verification was to embed the cache model in a larger model of the entire Leopard-2 system, and to include instrumentation within the larger model to monitor coherence states of memory lines. The instrumentation checks to ensure that coherence is maintained for each line.

4.3.1 The Leopard-2 System Model

At the top level of abstraction, the hardware model of the Leopard-2 Multiprocessor system, illustrated in Figure 4-1, comprises three GDP processor components and a shared memory component interconnected with a Futurebus backplane bus. Since the focus of the hardware model is on the caches and cache coherence, the I/O processor and controller components are not included. The number of GDPs was chosen to allow simulation of complex interactions between multiple processors sharing a line. An example scenario involves one cache suffering a write-miss and fetching a line, a second cache intervening to supply the line, and a third cache invalidating its copy of the line.

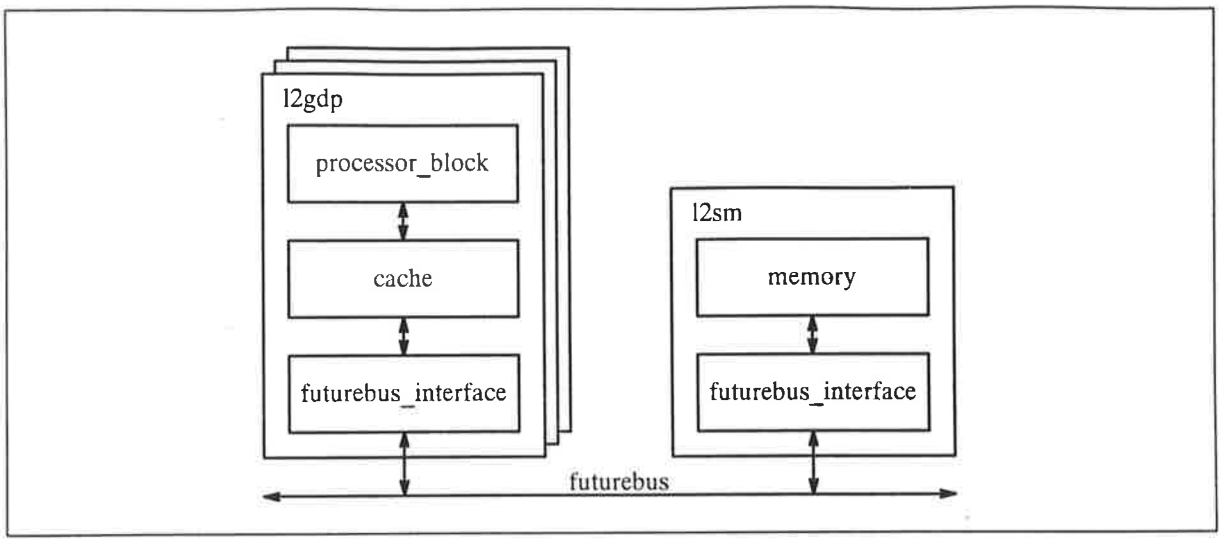


Figure 4-1. The top-level structure of the Leopard-2 hardware model.

The GDP is decomposed into a processor block component, a cache and a Futurebus interface. The shared memory is similarly decomposed into a memory component and a Futurebus interface.

The data values stored and transferred within the real Leopard-2 system are not directly modelled. Instead, each line of data is represented by a token that indicates the processor that most recently updated the line and the simulation time of the update. Processors are identified using their Futurebus geographic address, with the reserved value 0 indicating that the line has not been updated since initialization. The timestamps in the data tokens are used by the coherence monitoring instrumentation in the model to verify that each processor sees the latest version of the line.

The connections upon which address and data flow are further abstracted from their real electrical implementation. In the real hardware, buses are driven by tristate drivers, allowing different components to act as sources for an individual bus at different times. If no driver for a bus is enabled, the bus "floats", and any component sensing the bus receives an undefined value. Detailed electrical models represent this behaviour by augmenting the binary data type with a third value, "Z", indicating the high-impedance state of a tristate driver, and a fourth value, "X", indicating an unknown value. Since the model described here uses abstract tokens for data values and abstract integers for addresses, the augmented binary representation is inappropriate. Instead, buses are driven with a tuple comprising a valid flag and the abstract value. if the valid flag is false, the abstract value part is ignored; if the valid flag is true, the abstract value part is the value driven on the bus. The bus resolution function, which combines contributions from all drivers to determine the actual value on the bus, verifies that at most one driver is active at a time. This provides a significant degree of error checking in the model, and ensures that corrupted addresses and data values are not propagated without being detected.

The timing of interactions within each GDP are regulated by a clock signal generated by the processor block component. This models the processor clock in the real hardware. The connections between the processor block and the cache is an abstraction of the NS32532 bus interface. It includes just those control and status signals necessary to simulate transactions with the cache. The timing of the interactions, however, is directly modelled on the timing requirements of the NS32532 bus interface, referenced to the processor clock. Thus, for example, transactions are divided into sequences of timing states, with addresses issued and control signals activated at specified clock edges, and data required to be valid within specified numbers of clock cycles. Status signals are returned by the cache to allow it to insert wait cycles when data may be delayed.

Address and data values on the Futurebus are modelled using the same abstract types as are used within the GDP and memory components. However, the synchronization signals used for arbitration and data transactions are modelled in detail. Each data transaction is composed of three "beats": an address beat, during which address and command information are exchanged; a data beat, during which a data line token is transferred; and an end beat, during which completion status information is exchanged. In the real hardware, there are separate data beats for each word of data within a block. In the model, these are all collapsed into a single data beat for transfer of the data token.

The reason for modelling synchronization of arbitration and data transaction in detail is that the Futurebus protocols are asynchronous (not clocked) and involve synchronized interaction of several bus modules in different capacities such as master, slave and third party. There is no centralized component that manages synchronization; rather synchronization is fully distributed amongst those modules taking part in a transaction. If the Futurebus synchronization protocol were not used, then some other, equally complex, protocol would be required. It appeared best to use the existing proven protocol to synchronize the modelled components in the same way as the real components would be synchronized. Since the synchronization aspects of cache coherence protocols are the most difficult parts to implement correctly, detailed modelling of synchronization is important for verifying correctness of the cache design.

4.3.2 Workload Modelling in the Processor Block

The Leopard-2 system model is stimulated by read and write requests generated by the processor block on each L2GDP. These requests represent the workload performed by the Leopard-2 system. Two versions of the processor block component were developed, each generating requests differently.

The first processor block version generates requests according to a command file. The command file allows specification of synthetic address traces to exercise specific behaviours within the model. For example, during development of the cache model, synthetic traces were constructed to cause hits and misses within the cache. The detailed behaviour of the cache controller and cache datapaths was observed to verify that the correct operations were performed. The particular command file used by the processor block is specified by a generic parameter, and so can be varied for each instance within the system. This allows use of trace sets in which each processor performs different sequences of requests. For example, trace sets can be constructed to take a shared memory line through different coherence states in different processors, exercising particular aspects of the implementation of a cache coherence protocol.

The command files read by the first version of the processor block consist of lines of text, each specifying a request to be performed. An example trace file is

```
@ 1 us      R C  00000000       miss, sec 0 index 0
            W C  00000000       hit,  sec 0 index 0
            R C  00040000       miss, sec 1 index 0
            R C  00080000       miss, sec 0 index 0, copyback, flush buffer
            W N  00008000       write to buffer
            W N  00008004       write to buffer
            R C  00000040       miss, sec 0 index 1, flush buffer
+ 1 us      R C  00000000       miss, sec 1 index 0
```

The "@" symbol denotes an absolute simulation time at which the request is to be performed. The "+" symbol denotes a delay after completion of the previous request before the specified request is performed. The omission of a time denotes that the request is to be performed immediately after the previous request. Each request may be either a read (denoted by "R") or a write (denoted by "W"), and may be cachable ("C") or non-cachable ("N"). The address is specified in hexadecimal. Text following the address is a comment, and may be used to document the actions expected in response to the request.

The second version of the processor block generates read and write requests according to a programmed workload model. Previously published simulations of caches use workload models that attempts to mimic the behaviour of real workloads, since the aim of those simulations is to evaluate performance of the simulated system. Some of the models are discussed in Section 2.5. The aim of the simulation model discussed here, however, is to verify correctness of the cache design and implementation of cache coherence protocols, so the workload model is quite different. It is designed to cover randomly all cases of interactions between caches accessing a collection of shared memory lines. The approach taken is similar to that described by Wood *et al* [55] for testing the SPUR cache controller. Only a small number of lines are referenced, some being shared between processors, and others being referenced by only one processor. These two categories model shared and private data respectively. The lines are allocated in the address space as follows:

Shared:	00000000 – 000000FF
	00040000 – 000400FF
	00080000 – 000800FF
	000C0000 – 000C00FF
Private:	0000*nn*00 – 0000*nn*FF
	0004*nn*00 – 0004*nn*FF
	0008*nn*00 – 0008*nn*FF
	000C*nn*00 – 000C*nn*FF

The symbol *nn* denotes the geographic address of a GDP—each GDP uses its geographic address to determine the shared memory addresses of its private lines. Lines are 64 bytes each, so each group of consecutive addresses corresponds to four consecutive lines. Since there are 256 Kbytes of storage in each section of each cache, all of the groups of four lines map to sets 0 to 3 of each cache.

The algorithm for the workload model involves generating successive requests as follows:

- randomly choose a line to access, with probability of choice uniformly distributed amongst the shared and private lines accessible by the processor,

- randomly choose between a read request (with probability 0.75) and a write request (with probability 0.25)

Each processor uses its geographic address to calculate the seed for its pseudo random sequence generator used to make the random choices. If this were not done, each processor would generate the same sequence of memory requests, defeating the aim of the workload model.

As Wood *et al* note [55], the state space of interactions between caches is sufficiently large that manual generation of tests is intractable. While random testing makes testing feasible, it is not clear how good the test coverage is, nor how to analytically determine the level of coverage. Wood *et al* suggest this issue as a future research area. They do, however, report success in uncovering a number of design defects in their controller through use of randon testing.

Both versions of the processor block described above generate read and write requests on signals connected to the cache component. Timing of values on these signals is synchronized by the clock signal generated by the processor block, as shown in Figure 4-2. The processor issues the memory address, command information, and, for write requests, data during the T1 timing state, and pulses the address-strobe signal to start the access. If there is no delay in servicing the request, the cache responds by supplying status information and, for read requests, the data during the T2 state, and asserting the ready signal. If there is a delay, the cache leaves the ready signal negated, causing the processor block to insert wait states until ready is asserted. The status information that the cache returns includes a retry signal that is used to force the processor to abort the request and retry it. This is used when the cache receives a busy-retry response from the Futurebus while servicing a cache miss. The request timing and protocol described here is a simplification of that used by the NS32532, focussing on just those aspects needed to model the coherent caching behaviour of the Leopard-2 system.

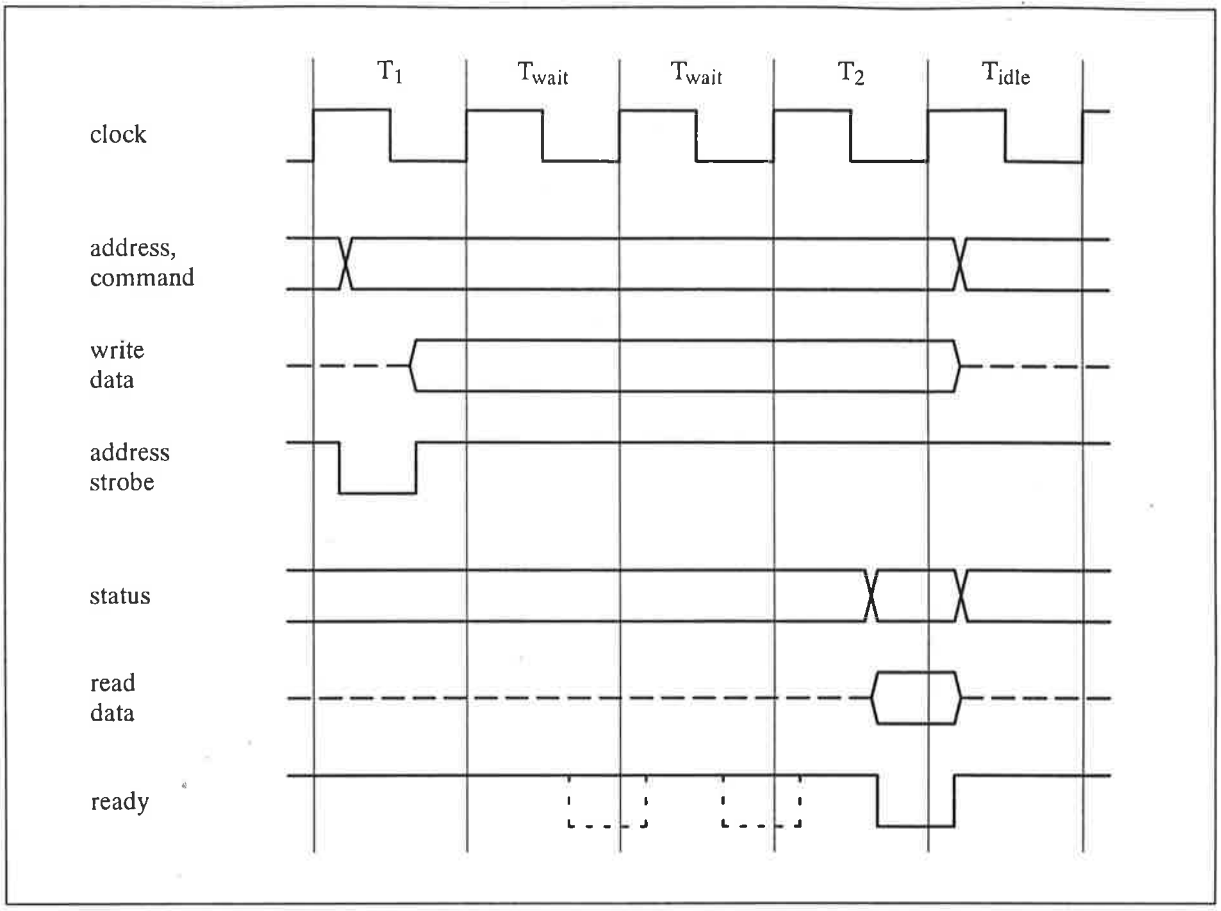


Figure 4-2. Timing of processor read and write requests.

4.3.3 The Cache Model

The cache subsystem of the Leopard-2 GDP consists of a datapath, described in Section 3.7.2, and a cache controller, which operates as described in Section 3.8. This organization is mirrored in the hardware model of the cache subsystem. The structure of the datapath is modelled at the register transfer level, with separate component instances for each of the elements in the datapath. The simpler elements, such as transceivers and comparators, are modelled using a dataflow style. More complex elements are modelled using a behavioural style, in which the function of the element is expressed in the form of an algorithm using processes containing sequential statements.

```
type parameter_set is record
        excl_depends_on_CS_on_read_shared : boolean;
        tr_write_hit_shared : transaction_type;
        owned_on_write_hit_shared : boolean;
        excl_depends_on_CS_on_write_hit_shared : boolean;
        tr_write_miss : transaction_type;
        reflect_on_read_shared : boolean;
        inval_if_third_party : boolean;
        sel_on_broadcast_hit : boolean;
    end record parameter_set;
```

Figure 4-3. The record type used to define cache parameter values.

Operation of the datapath elements is controlled by control signals, sequenced by the cache controller synchronized by the clock signal from the processor block. In the real Leopard-2 hardware, the sequence of operations would be devised to take as few clock cycles as possible, to maximize performance. While the hardware model pays significant attention to reducing the number of clock cycles for each sequence, there may be scope for further optimization. The main aim in developing the model was correct operation; optimal sequencing was beyond the scope of the design.

The cache controller model is parameterized using the scheme described in Section 4.2. The entity interface for the controller includes a generic constant for specifying the particular parameter values to be taken on by a cache controller instance. The generic constant is a record of the type shown in Figure 4-3. The modelled cache is programmed to implement a particular coherence protocol by supplying actual parameter values for the generic constant according to Table 4-2.

The cache controller model is behavioural in style, and consists of a number of communicating processes, illustrated in Figure 4-4. The master process sequences cache actions in response to processor read and write requests, the snoop process sequences cache actions in response to transactions observed on the Futurebus, and the write

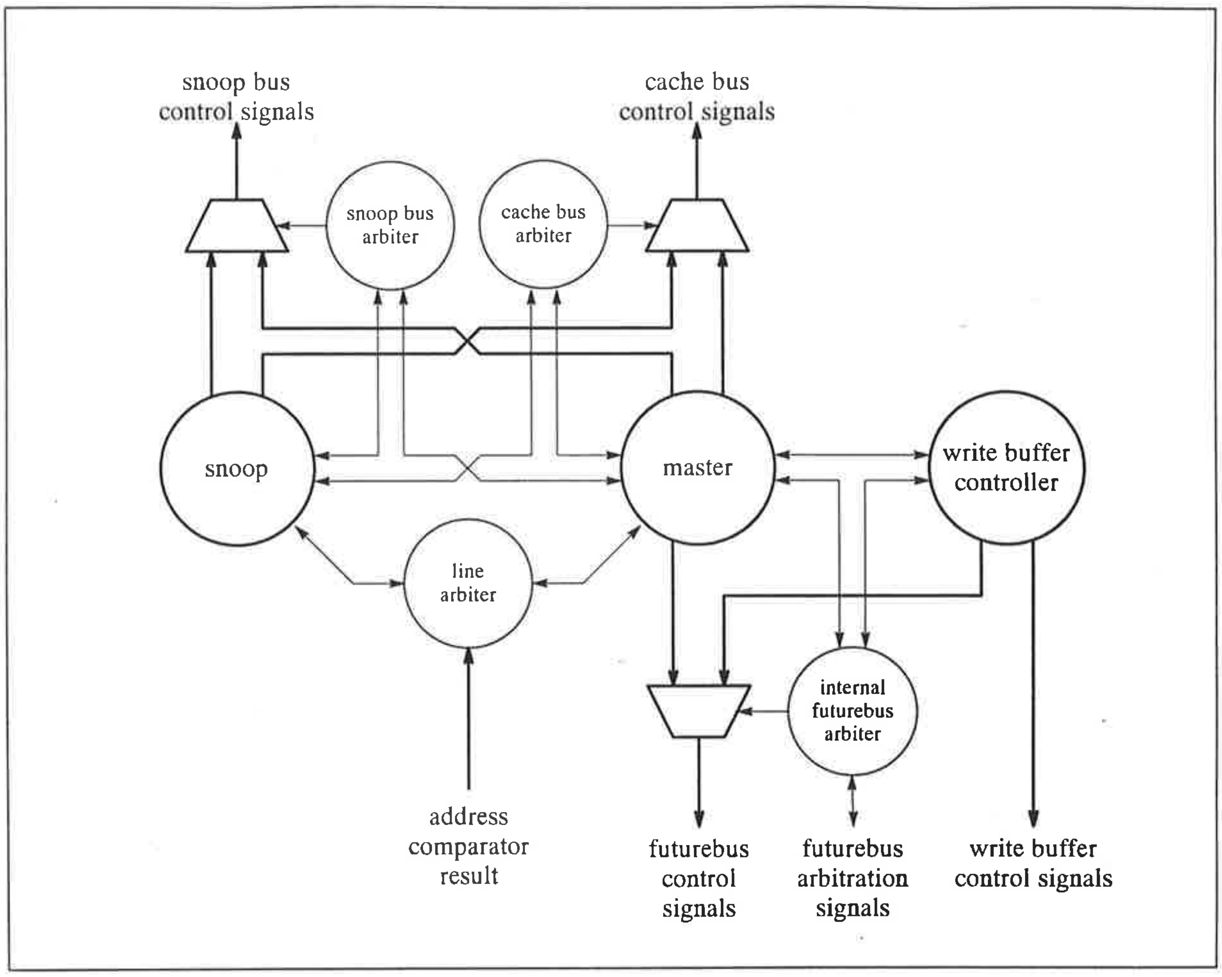

snoop bus control signals
cache bus control signals
snoop bus arbiter
cache bus arbiter
snoop
master
write buffer controller
line arbiter
internal futurebus arbiter
address comparator result
futurebus control signals
futurebus arbitration signals
write buffer control signals

Figure 4-4. The process structure of the cache controller model.

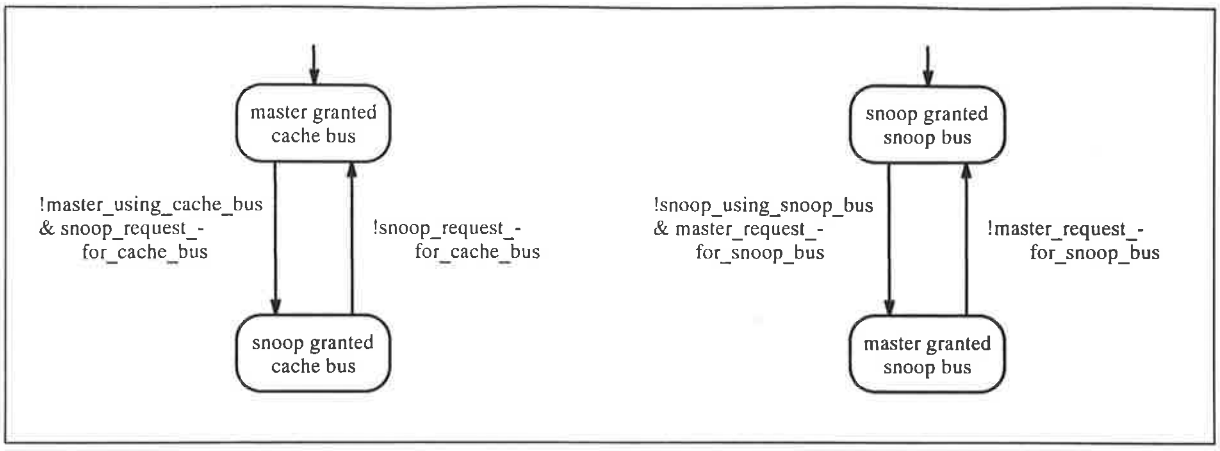
buffer process sequences flushing to shared memory of write data from the write buffer. The remaining processes are arbiters and multiplexers, discussed below.

Cache Controller Arbiters

Many of the control signals for various parts of the cache datapath must be sequenced by different processes at different times. For example, control signals for the snoop bus must be sequenced by the snoop process when a Futurebus transaction is in progress, but must be sequenced by the master process when the snoop's copy of attributes is being updated. The requirement for shared control leads to the inclusion of an arbiter for each set of shared resources: one for the cache buses shared by the master and snoop processes, one for the snoop buses shared by the master and snoop processes, and one for the Futurebus interface shared by the master and write buffer processes. Associated with each arbiter is a multiplexer that selects control signal values from the process that is granted access to the set of resources.

The remaining arbiter in the cache controller is used to ensure mutual exclusion between the processor and the snoop operating on lines of the shared memory address space. The requirement for mutual exclusion is discussed in Section 2.2.11, and its implementation in the cache controller algorithm is described in Section 3.8. The line arbiter takes as input the result of the address comparator in the cache datapath. This comparator compares the line address being accessed as part of servicing the processor request with the line address being accessed by the snoop. If both the processor and the snoop are active and the line addresses are equal, there is contention for use of the line, so the processor and snoop operations are serialized. Otherwise, if only one of the processor or snoop is active or if they accessing different lines, there is no contention, so they may perform their operations concurrently.

The presence of the arbiters and the serialization that they enforce are manifest in interference between the processor and the snoop. While interference due to mutual exclusion over line addresses is expected to be relatively rare (except in cases of heavily shared data), interference due to shared access to cache and snoop buses is expected to be relatively more frequent. (The actual frequency depends on the access patterns of the particular code being run on the system.) This is because it occurs whenever the processor or snoop must update attributes of a line, irrespective of which line the other process is accessing, or when the snoop must access the cache data memory to act as a third party or to receive a broadcast update. Fortunately, the duration of each instance of these latter cases of interference is less than that of interference due to mutual exclusion over lines, hence its effect on performance is minimal. The performance effects are further ameliorated by allowing the processes to proceed speculatively on the


master granted cache bus
!master_using_cache_bus & snoop_request_-for_cache_bus
!snoop_request_-for_cache_bus
snoop granted cache bus
snoop granted snoop bus
!snoop_using_snoop_bus & master_request_-for_snoop_bus
!master_request_-for_snoop_bus
master granted snoop bus
Figure 4-5. State transition diagrams for the cache bus arbiter (left) and snoop bus arbiter (right).

assumption of being granted access to a resource immediately upon requesting it, and blocking later if they find the resource denied.

The arbiters are implemented as clocked finite state machines. The cache bus and snoop bus arbiters are the simplest. Their state transition diagrams are shown in Figure 4-5. Normally the master is granted access to the cache bus, so there is no explicit request from the master. Instead, the master indicates when it is actually using the cache bus. The snoop has an explicit request signal, and is only granted the cache bus when the master is not using it. While the snoop is granted the cache bus, the master refrains from using it. When the snoop has finished using the cache bus, it removes its request, allowing the master to proceed when it needs the bus. Operation of the snoop bus arbiter is similar, but with the snoop normally granted access and the master having to request access.

The state transition diagram for the line arbiter is shown in Figure 4-6. In addition to the transition conditions shown in the diagram, transitions are only taken when neither master is granted use of the snoop bus nor snoop is granted use of the cache bus. This condition is required to prevent false transitions due to address matches when an address from one side (master or snoop) is transmitted to the address bus of the other side. The condition is omitted from the diagram for clarity.

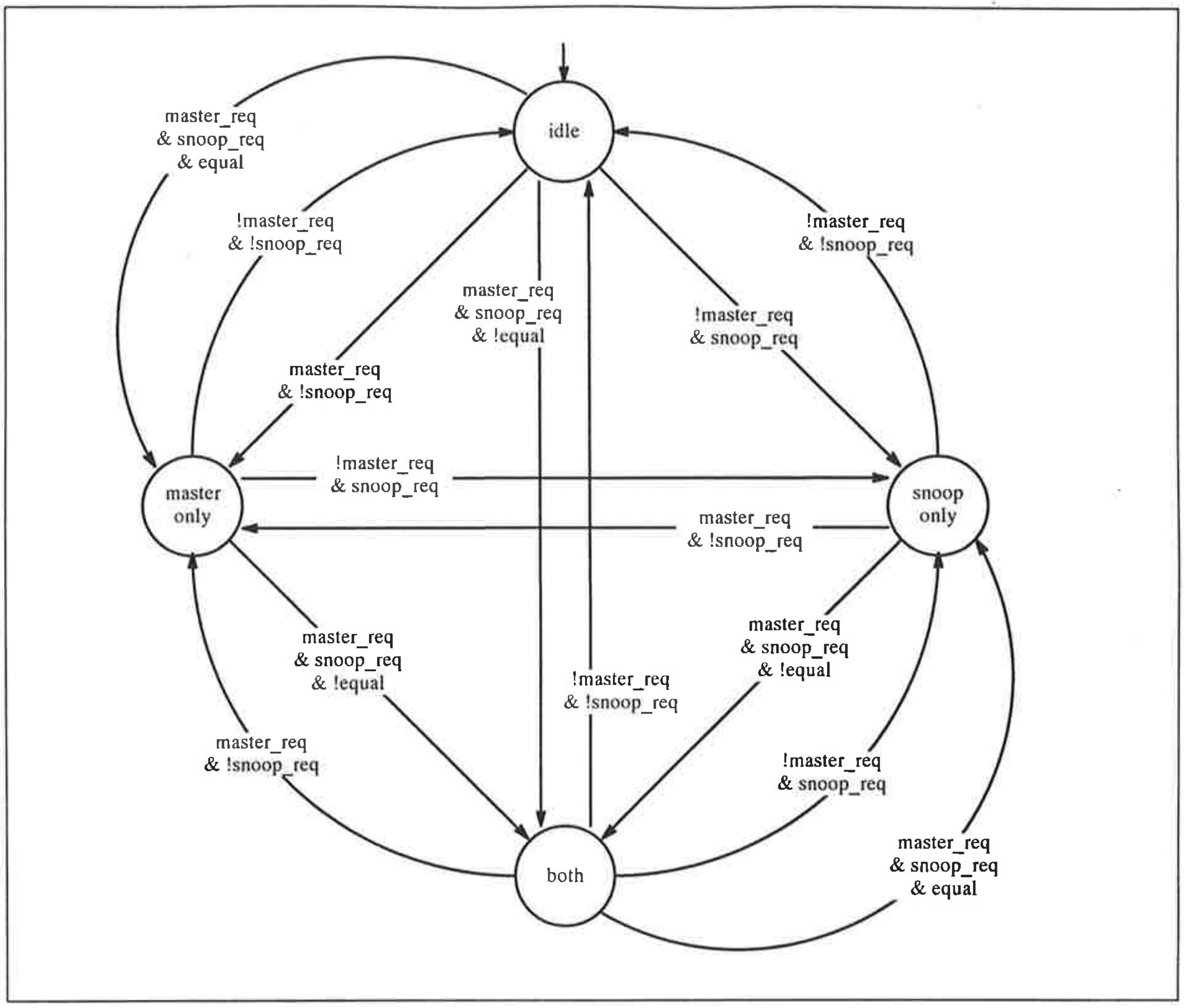


Figure 4-6. State transition diagram for the line arbiter.

Initially the arbiter is in the "idle" state with neither master nor snoop active. If the master alone requests use of a line, the arbiter transitions to the "master only" state, granting permission for the master to use the line. Similarly, if the snoop alone requests use of a line, the arbiter transitions to the "snoop only" state, granting permission for the snoop to use the line. If both master and snoop simultaneously request use of lines, and the line addresses are not equal (denoted by "!equal" in the diagram), the arbiter transitions to the "both" state and both are granted permission to proceed. If both requests are made simultaneously but the line addresses are equal, the arbiter grants permission to the master first. Only when the master completes its request, or when the master's line address changes, is the snoop allowed to proceed. When the arbiter is in the "both" state, the line addresses may change from being unequal to being equal. This may occur when the master must replace a line and move it to the write buffer. Whilst moving the replaced line, the mutual exclusion lock must be applied over the address of the replaced line, to prevent the snoop trying to perform coherence transaction on the line while it is not available. (Once the replaced line is in the write buffer, the copyback address comparator is used to prevent further coherence transactions on the line until it has been written back to shared memory.) At the start of the move, the master's line address changes from that of the line requested by the CPU to that of the replaced line. At the end of the move, the master's line address reverts to that of the line requested by the CPU. In both cases, if the snoop is currently granted use of the line to which the master's address changes, the snoop is allowed to complete its activity before the master is allowed to proceed.

The state transition diagram for the internal Futurebus arbiter is shown in Figure 4-7. The function of this arbiter is to merge requests for the Futurebus from the cache master and the write buffer. The merged request is forwarded to the Futurebus arbiter in the Futurebus interface, and the returned grant is directed back to the cache master or write buffer as appropriate. A complication is that the cache master may preempt the write buffer. If the write buffer has acquired the Futurebus to perform queued non-cachable writes, and the cache master needs to satisfy a read miss, it preempts the write buffer in order to read the required line without delay, then returns access to the Futurebus to the write buffer.

When the cache master requests the Futurebus, the request is forwarded to the Futurebus interface, and when the bus is granted, the grant is returned to the cache master. A write buffer request is only forwarded to the Futurebus interface if there is no concurrent cache master request. If, by the time the bus is granted, a cache master request has arrived, the grant is forwarded to the cache master in preference to the write buffer.

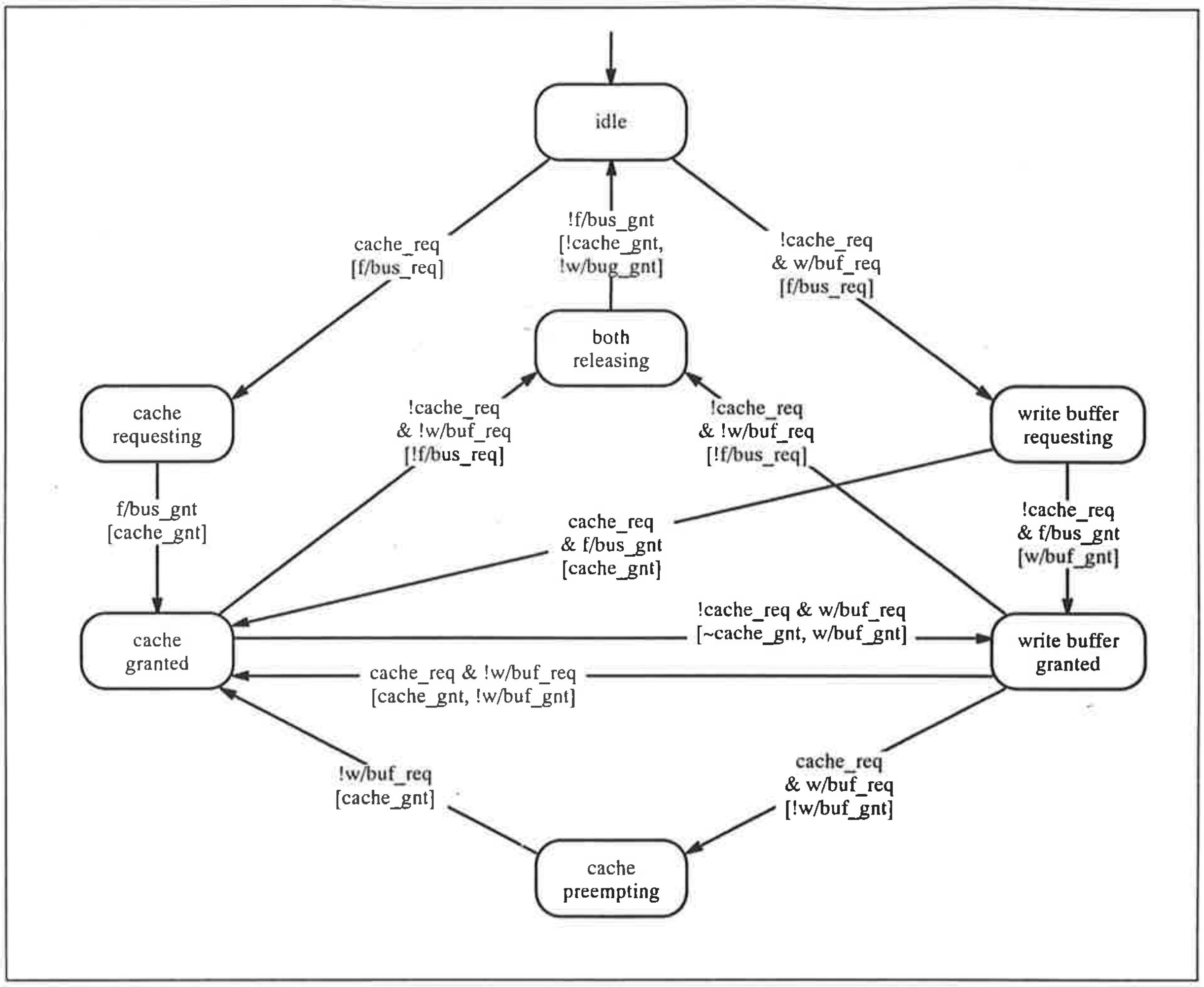

idle
cache_req [f/bus_req]
!f/bus_gnt [!cache_gnt, !w/bug_gnt]
!cache_req & w/buf_req [f/bus_req]
both releasing
cache requesting
!cache_req & !w/buf_req [!f/bus_req]
!cache_req & !w/buf_req [!f/bus_req]
write buffer requesting
f/bus_gnt [cache_gnt]
cache_req & f/bus_gnt [cache_gnt]
!cache_req & f/bus_gnt [w/buf_gnt]
!cache_req & w/buf_req [~cache_gnt, w/buf_gnt]
cache granted
write buffer granted
cache_req & !w/buf_req [cache_gnt, !w/buf_gnt]
!w/buf_req [cache_gnt]
cache_req & w/buf_req [!w/buf_gnt]
cache preempting

Figure 4-7. State transition diagram for the internal Futurebus arbiter.

When both requests are negated, the Futurebus request is also negated, and when the Futurebus grant is negated by the Futurebus interface, the returned grant to the cache master or write buffer is also negated. If a write buffer request arrives at the same time as the cache master request is removed, tenure of the Futurebus is maintained, and the grant is transferred from the cache master to the write buffer. Tenure is transferred from the write buffer to the cache master in the case of a cache master request arriving at the same time that the write buffer request is removed. The remaining transitions implement preemption of the write buffer by the cache master, signalled by a cache master request arriving while the bus is granted to the write buffer. In this case, the grant is removed from the write buffer. When the write buffer has completed its current bus transaction, it removes its request. This signals to the arbiter that bus tenure can be transferred to the cache master. The write buffer must then re-assert its request to regain bus tenure after the cache master has completed its transactions.

Master Sequencer Outline

The master process shown in Figure 4-4 models the sequencing of cache datapath operations in response to read and write requests from the processor. The process uses the clock signal generated by the processor to synchronize operations. A pseudo-code outline of the master process is shown in Figure 4-8. The process waits until a processor request arrives (middle of a T_1 clock cycle, on the falling clock edge), then decodes the command information to determine how to respond. In the case of non-cachable reads, a procedure is called to sequence the required Futurebus read transaction. In the case of non-cachable writes, a procedure is called to write the data into the write buffer. The details of these two procedures is not discussed here, since the focus is on coherent cachable memory accesses.

If the processor request is for a cachable accesses, the master process calls a procedure to sequence looking up the cache to check for a hit. This takes until the falling clock edge of the subsequent clock cycle (T_2 or T_{wait}). If the access is found to miss, a procedure is called to sequence the miss. This will take a number of T_{wait} clock cycles, returning on the falling clock edge of a cycle with the required line having been fetched into the cache, or with a retry status, requiring the processor to abort and retry the request. The procedure that sequences aborting with retry simply returns the cache control signals to their quiescent state, releases resources acquired by the master through arbitration, and responds to the processor with retry status. In the case of the line being successfully fetched, the process simply falls through into the section that sequences a hit. For a read hit, the master process calls a procedure to sequence accessing the

```
master : process is
begin
    CPU_transaction_loop : loop

        wait until falling clock edge in T1

        if cpu_command is non-cachable read then
            sequence_non_cachable_read
        elsif cpu_command is non-cachable write then
            sequence_non_cachable_write
        else
            sequence_cache_lookup
            if not hit then
                sequence_miss
                if abort_with_retry then
                    sequence_abort_with_retry
                end if
            end if
            if not abort_with_retry then
                if cpu_command is read then
                    sequence_read_hit
                else
                    sequence_write_hit
                    if abort_with_retry then
                        sequence_abort_with_retry
                    end if
                end if
            end if
        end if

    end loop CPU_transaction_loop
end process master
```

Figure 4-8. Outline of the master process.

cache for the required data. For a write hit, the process calls a procedure that sequences updating the cache, and performing any Futurebus transaction required to maintain coherence. Since such a Futurebus transaction may return retry status, the write sequencing procedure may also return retry status. Upon completion of sequencing the processor's request, the master process loops and waits for the next request.

```
procedure sequence_cache_lookup is
begin
    master_requesting_line <= true

    enable cpu tag latch and set index latch outputs
    enable cpu tag ram comparison
    enable cpu attribute ram outputs
    cache_offset <= cpu_offset  for data  ram address

    loop
        wait until rising edge of clock
        exit when snoop not using cache bus and master granted mutex to line
    end loop
    master_using_cache_bus <= true

    wait until middle of clock cycle

    sample current cpu attributes
    sample current lru section
    assume initially hit is false
    for each section loop
        if tag match and cpu attribute fb_valid is set then
            hit is true, hit_section_index is this section
        end if
    end loop

    disable cpu tag ram comparison
    disable cpu attribute ram outputs

end procedure sequence_cache_lookup
```

Figure 4-9. Outline of the procedure that checks for a cache hit.

The procedure that checks for a cache hit is shown in Figure 4-9. The procedure is called on the falling clock edge of the T_1 cycle of the processor's request. It immediately requests mutually exclusive access to the line by asserting its arbitration request signal. In fact, the arbitration request signal passed to the line arbiter is the logical-or of the processor's address strobe signal and the request generated by the master process. Since the line arbiter is clocked on falling edges of the processor clock, this scheme allows arbitration to commence during the T_1 cycle rather than being delayed until the next cycle. Furthermore, in the common case of the master being granted mutual exclusion for the requested line immediately, the result of arbitration is known in time for the master to proceed without delaying the processor.

The procedure in Figure 4-9 must also arbitrate for access to the cache buses and control signals, since the snoop may be using them. Rather than waiting for the outcome of this arbitration, the procedure speculatively activates its control outputs in the expectation that it already has won arbitration. (As described earlier, the master is deemed always to be requesting use of the buses and control signals, so is granted access as long as the snoop has not already acquired them.) The arbiter is clocked on the rising edge of the processor clock, so the result of arbitration is known at the beginning of the next clock cycle. The procedure waits until granted access before proceeding. In the common case, the master is granted access immediately, and can proceed without delaying the processor.

The control outputs that are speculatively enabled cause the CPU's address to be placed on the cache address bus, the tag RAM to use this address to compare with the stored tags, the CPU attributes to be read, and the data RAM to start accessing the set of data. When use of the cache buses is available and the master has mutual exclusion over the requested line, the procedure delays until the falling clock edge of of the next clock cycle, then samples the CPU attributes, LRU section number and tag comparison results. It uses the attributes and comparison results to determine if the requested line is hit in the cache. Having done so, it disables the tag RAM and CPU attributes RAM. The CPU address is left enabled for subsequent steps. Upon return from the procedure, the request has progressed to the falling clock edge of the cycle after T_1, and the master has determined whether the requested line is present, and if so, in which section.

The procedure that sequences a read hit is shown in Figure 4-10. The procedure enables the data RAM and transceivers to supply data to the CPU, and asserts the ready signal to allow the CPU to proceed. It then enables the CPU attributes RAM to write back the CPU attributes unchanged and to update the LRU section number to refer to

```
procedure sequence_read_hit is
begin
    enable data RAM hit section output
    enable cpu/cache data transceiver in cache-to-cpu direction

    assert CPU ready signal

    cpu_attributes(hit_section_index) <= current_cpu_attributes(hit_section_index)
    lru_section <= new_lru_section
    write-enable hit_section_index of cpu attributes RAM (also writes lru_section)

    wait until rising clock edge at end of T2 clock cycle

    negate CPU ready signal
    disable cpu tag latch and set index latch outputs
    cache_offset <= undriven
    disable cpu/cache data transceiver
    disable cpu attributes RAM
    disable data RAM
    cpu_attributes(hit_section_index) <= undriven
    lru_section <= undriven

    master_requesting_line <= false
    master_using_cache_bus <= false
end procedure sequence_read_hit
```

Figure 4-10. Outline of the procedure that sequences a read hit.

the other of the two sections, and asserts the write-enable control signal to the CPU attributes RAM. The procedure then waits until the rising clock edge at the end of the clock cycle, then disables all of the active control signals, releases the mutual exclusion lock on the line, and makes the cache bus accessible to the snoop.

The procedure that sequences a write hit is shown in Figure 4-11. The procedure first checks whether the line is shared, and if so, calls a procedure to sequence the Futurebus transaction required to maintain coherence. If it returns with retry status, the write-hit procedure also returns immediately with retry status. If it returns indicating that the write hit should be transformed into a miss (due to the snoop invalidating the line while the master was waiting for the Futurebus), the write-hit procedure calls a proced-

```
procedure sequence_write_hit is
begin
    if cpu_attributes(hit_section_index).fb_exclusive is clear then
        sequence_write_hit_to_shared_transaction
        if abort_with_retry then
            return
        end if
        if turn_hit_into_miss then
            sequence_read_for_miss
                using hit_section_index as replaced_section_index
            if abort_with_retry then
                return
            end if
            if cpu_attributes(hit_section_index).fb_exclusive is still clear then
                sequence_write_hit_to_shared_transaction
                if abort_with_retry then
                    return
                end if
            end if
        end if
    end if

    if cpu_attributes(hit_section_index).fb_exclusive is clear then
        set/clear required_fb_owned_value depending on
            owned_on_write_hit_shared
        set/clear required_fb_excl_value depending on
            excl_depends_on_CS_on_write_hit_shared
            (use saved cache_status value if excl_on_write_hit_shared is excl)
    else
        set required_fb_owned_value and required_fb_excl_value
    end if

    if required_fb_owned_value differs from current fb_owned value
        or required_fb_excl_value differs from current fb_excl value then

        master_request_for_snoop_bus <= true

        update current_cpu_attributes(hit_section_index)
            with required attribute values
        cpu_attributes(hit_section_index)
            <= current_cpu_attributes(hit_section_index)
        lru_section <= new_lru_section
        enable cpu/snoop address transceiver in cpu-to-snoop direction
        snoop_attributes(hit_section_index)
            <= current_cpu_attributes(hit_section_index)

        wait until rising clock edge  and master_granted_snoop_bus
        . . .
```

Figure 4-11. Outline of the procedure that sequences a write hit.

```
            . . .
            write enable hit_section_index of cpu attributes RAM (also writes lru section)
            write enable hit_section_index of snoop attributes RAM

            wait until falling clock edge

            master_requesting_snoop_bus <= false

            cpu_attributes(hit_section_index) <= undriven
            lru_section <= undriven
            disable cpu attributes RAM
            disable cpu/snoop address transceiver
            snoop_attributes(hit_section_index) <= undriven
            disable snoop attributes RAM
        end if

        enable cpu/cache data transceiver in cpu-to-cache direction
        write-enable hit_section_index of data RAM
        assert CPU ready signal

        wait until rising clock edge at end of T2 clock cycle

        negate CPU ready signal
        disable cpu tag latch and set index latch outputs
        cache offset <= undriven
        disable cpu/cache data transceiver
        disable data RAM

        master_requesting_line <= false
        master_using_cache_bus <= false
    end procedure sequence_write_hit
```

Figure 4-11 (continued).

ure to read the missed line, then if the line is still shared, calls the write-hit transaction procedure again. That procedure will complete normally, since Futurebus tenure is maintained between the read for the miss and the write-hit transaction, so there is no opportunity for the snoop to invalidate the line a second time.

Next, if the line is shared, the write-hit procedure determines new values for the fb_owned and fb_exclusive attributes of the line, based on the values of the configuration parameters owned_on_write_hit_shared and excl_depends_on_CS_on_-

write_hit_shared. In the case of a write hit to an exclusive line, fb_owned is alway set, and the fb_exclusive attribute remains set.

The write hit procedure next determines whether the stored attributes must be updated. It compares the required attribute values with the current values, and if there is a difference, the update proceeds. The procedure arbitrates for access to the snoop bus, and sets up the new attribute values and LRU section index for the CPU and snoop attribute RAMs. When access to the snoop bus is granted, the CPU and snoop attribute RAMs are updated. The procedure then releases the snoop bus and disables all control signals used to update the attribute RAMs. The write-hit procedure next sequences the local write to the cache. It enables the data transceivers to forward data from the CPU to the cache, enables the data RAM to accept the data from the CPU, and asserts the ready signal to allow the CPU to proceed. The procedure then waits until the rising clock edge at the end of the clock cycle, then disables all of the active control signals, releases the mutual exclusion lock on the line, makes the cache bus accessible to the snoop.

The procedure that sequences a Futurebus transaction on a write hit to a shared line is shown in Figure 4-12. If the master does not currently have Futurebus tenure, the procedure first releases the mutual exclusion lock on the line and makes the cache buses available to the snoop. It then requests arbitration for the Futurebus. When access to the Futurebus is granted, the procedure reacquires the mutual exclusion lock on the line and use of the cache buses. It need not wait for these to be granted; since the master has Futurebus tenure, the snoop and all other masters in the system must be inactive. Next, the procedure re-reads the attributes from the CPU attribute RAM, since the line may have been invalidated by the snoop while the master was waiting for the Futurebus. If the line is invalid, the procedure returns with a status value indicating that the write hit should be turned into a write miss.

```
procedure sequence_write_hit_to_shared_transaction is
begin
    if  not cache_grant then
        master_requesting_line <= false
        master_using_cache_bus <= false

        cache_request <= true
        wait until falling clock edge and cache_grant

        master_requesting_line <= true
        master_using_cache_bus <= true

        enable cpu attribute RAM outputs
        wait until middle of clock cycle
        sample current cpu attributes
        disable cpu attribute RAM outputs

        if fb_valid attribute in hit section is now clear then
            turn_hit_into_miss := true
            return
        end if
    end if

    if tr_write_hit_shared is a write transaction then
        enable cpu/cache data transceiver in cpu–to–cache direction
    else
        disable cpu/cache data transceiver
    end if

    if write buffer is not empty then
        flush_write_buffer <= true
        wait until falling clock edge and write buffer is empty
    end if
    flush_write_buffer <= false

    enable cache/Futurebus address and data transceivers
    data direction depends whether tr_write_hit_shared is a read or write transaction

    assert Futurebus cache_command and intent_to_modify
    assert/negate Futurebus broadcast depending on tr_write_hit_shared
    assert/negate Futurebus write depending on tr_write_hit_shared
    assert/negate Futurebus ownership depending on owned_on_write_hit_shared

    assert Futurebus address strobe
    wait until falling clock edge and Futurebus address acknowledge asserted
    save Futurebus cache_status reply value
    . . .
```

Figure 4-12. Outline of the procedure that sequences a Futurebus transaction on a write hit to a shared line.

```
    . . .
    if Futurebus busy reply then
        abort_with_retry := true
    elsif tr_write_hit_shared is not invalidate then
        assert/negate Futurebus three_party depending on third_party reply
        assert Futurebus data strobe
        if tr_write_hit_shared is a read transaction then
            write enable hit section of data RAM
        end if
        wait until falling clock edge and Futurebus data acknowledge asserted
        if tr_write_hit_shared is a read transaction then
            disable data RAM
        end if
        negate Futurebus data strobe
    end if
    negate Futurebus address strobe
    wait until falling clock edge
                and Futurebus address and data acknowledge both negated

    negate Futurebus_command
    disable cache/Futurebus address and data transceivers

    if tr_write_hit_shared is a read transaction then
        enable cpu/cache data transceiver in cpu-to-cache direction
    end if

    cache_request <= false
    wait until falling clock edge and not cache_grant

end procedure sequence_write_hit_to_shared_transaction
```

Figure 4-12 (continued).

If the transaction can proceed, its type is specified by the parameter tr_write_hit_shared. In the case of a write transaction being required, the procedure enables the CPU's data onto the cache data bus, otherwise it disables the CPU data transceivers. The procedure then signals the write buffer to drain, and waits until there are no entries in the write buffer. It prepares for the Futurebus transaction by enabling the transceivers between the cache buses and the Futurebus interface, and by setting up the Futurebus command values based on the configuration parameter values. The procedure then sequences the Futurebus transaction, starting with an address beat. It saves the value of the Cache-Status reply signal for later use in updating the attributes of the line. In addition, it saves the state of the busy reply signal for use after the bus transaction is complete. If there is no busy reply and the transaction is not an address-only invalidate, the transaction proceeds with a data beat, reading data into the data RAM or writing data from the cache data bus depending on the type of transaction. When the transaction sequence is complete, the procedure negates the Futurebus command values, disables the cache-to-Futurebus transceivers, re-enables the CPU-to-cache data transceiver, and releases tenure of the Futurebus.

The procedure that sequences a cache miss is shown in Figure 4-13. The procedure first selects a section in the current set for replacement, preferring an entry that is not valid if one exists, otherwise choosing the least recently used entry. It next examines the Dirty attribute of the replaced line to determine whether the line must be copied back via the write buffer. If so, the procedure releases the mutual exclusion lock on the line and use of the cache buses, in order to avoid deadlock with other caches via the snoop during subsequent operations. The procedure then waits until there is no previously buffered line pending in the write buffer and there is room for the newly replaced line. It then sets a flag to indicate to the write buffer sequencer that the line about to be placed in the write buffer is replaced as part of the current miss. This is required to prevent the write buffer from flushing that line before the line requested by the CPU is fetched from shared memory. The procedure next calls a procedure to sequence the transfer of the replaced line to the write buffer. The miss sequencing procedure then calls a procedure to sequence the read transaction to fetch the required line. Finally, it clears the flag allowing the write buffer to proceed with flushing the replaced line.

The procedure that sequences copying a dirty replaced line to the write buffer is shown in Figure 4-14. The procedure first replaces the tag field on the cache address bus with the tag of the replaced line, and sets the offset part of the address to 0. It then

```
procedure sequence_miss is
begin
    set replaced_section_index to lru_section
    for each section
        if cpu_attributes(section_index).fb_valid is clear then
            set replaced_section_index to this section
        end if
    end loop

    if cpu_attributes(replaced_section_index).dirty is set then
        master_requesting_line <= false
        master_using_cache_bus <= false
        if copyback pending or write buffer full then
            wait until falling clock edge
                        and no copyback pending and write buffer not full
        end if
        current_miss_required_copyback <= true
        sequence_copyback
    end if

    sequence_read_for_miss
    current_miss_required_copyback <= false
end procedure sequence_miss
```

Figure 4-13. Outline of the procedure that sequences a cache miss.

requests mutual exclusion for the replaced line. It initiates the re-reading of the CPU attributes, then waits until the mutual exclusion lock on the replaced line is granted and the snoop is not using the cache buses. When access is made available, the procedure claims use of the cache buses, then re-samples the CPU attributes and disables the CPU attribute RAM outputs.

The procedure next checks whether the replaced line is still valid and dirty. If it is, the line must still be copied back. The procedure sets a flip-flop that signals the presence of a replaced line pending copy-back in the write buffer, enables the data RAM output to supply the line, and sets up the write buffer marks. When the address, data and mark values are stable on the cache buses, the procedure enables the copy-back address

```
procedure sequence_copyback is
begin
    disable cpu tag latch output
    output enable replaced_section_index of cpu tag RAM
    cache_offset <= 0 for copyback address

    master_requesting_line <= true

    enable cpu attribute RAM outputs

    loop
        wait until rising clock edge
        exit when snoop not using cache bus and master granted mutex to line
    end loop
    master_using_cache_bus <= true

    wait until falling clock edge

    sample current cpu attributes
    disable cpu attribute RAM outputs

    if fb_valid and dirty attribute of replaced_section_index are both set then
        set copyback_pending flip–flop
        enable data RAM replaced section output
        write through mark <= '0'
        cache command mark <= '0'

        wait until rising clock edge at start of next clock cycle

        enable write to copyback address comparator
        enable shift–in to write buffer fifo

        wait until falling clock edge and write buffer fifo accepted input

        disable write to copyback address comparator
        disable shift–in to write buffer
        disable data RAM replaced section output
    end if;

    disable replaced_section_index of cpu tag RAM
    cache_offset <= cpu_offset

    master_requesting_line <= false
    master_using_cache_bus <= false

end procedure sequence_copyback
```

Figure 4-14. Outline of the procedure that copies a replaced line to the write buffer.

comparator and write buffer FIFO inputs. After a delay for the inputs to be accepted, the comparator and FIFO inputs are disabled and the data RAM output is disabled. This completed transfer of the replaced line to the write buffer. The final actions of the procedure are to restore the address of the line requested by the CPU on the cache address bus, to release the mutual exclusion lock on the replaced line, and to make the cache buses available to the snoop.

The procedure that sequences a read transaction on a cache miss is shown in Figure 4-15. The procedure is called either on an initial miss in response to a CPU request, or on a write miss resulting from a snoop invalidation of a shared line during a write hit. In the former case, the procedure is called without the master having Futurebus tenure, and the mutual exclusion lock on the line may or may not be held, depending on whether a copyback preceded the call. In the latter case, the procedure is called with the master having Futurebus tenure and the mutual exclusion lock on the line in place.

The procedure first checks whether the master has Futurebus tenure. If not, it ensures that the mutual exclusion lock and use of the cache buses are released, then requests arbitration for the Futurebus so that it can initiate the transaction required to read the line. When access to the Futurebus is granted, the procedure reacquires the mutual exclusion lock on the line and use of the cache buses. It need not wait for these to be granted; since the master has Futurebus tenure, the snoop and all other masters in the system must be inactive. The procedure then signals the write buffer to drain, and waits until there are no entries in the write buffer, or until the entry at the head of the write buffer is the replaced line transferred earlier in processing the current miss. The procedure next determines the required type of Futurebus transaction: read-shared for a read miss; or read-shared or read-invalidate, specified by the tr_write_miss parameter, for a write miss. The procedure prepares for the Futurebus transaction by enabling the transceivers between the cache buses and the Futurebus interface, and by set-

```
procedure sequence_read_for_miss is
begin
    if not cache_grant then
        master_requesting_line <= false
        master_using_cache_bus <= false
        cache_request <= true
        wait until falling clock edge and cache_grant

        master_requesting_line <= true
        master_using_cache_bus <= true
    end if

    if write buffer is not empty
        and ( write through mark output is set
                or pending copyback is from previous miss ) then
        flush_write_buffer <= true
        wait until falling clock edge
                    and ( write buffer is empty
                            or ( write through mark is clear
                                and pending copyback is from this miss ) )
    end if
    flush_write_buffer <= false

    if cpu request is read then
        set required_transaction to read_shared
    else
        set required_transaction to tr_write_miss parameter value
    end if

    enable cache/Futurebus address and data transceivers for read direction

    assert Futurebus cache_command
    negate Futurebus broadcast, three_party and write
    if required_transaction is read_invalidate then
        assert Futurebus intent_to_modify and ownership
    else
        negate Futurebus intent_to_modify and ownership
    end if

    assert Futurebus address strobe
    wait until falling clock edge and Futurebus address acknowledge asserted
    save Futurebus cache_status reply value
    if Futurebus busy reply then
        abort_with_retry := true
    else
        . . .
```

Figure 4-15. Outline of the procedure that sequences a read transaction on a cache miss.

```
    . . .
    assert/negate Futurebus three_party depending on third_party reply
    assert Futurebus data strobe
    write enable replaced section of data RAM
    **wait until** falling clock edge and Futurebus data acknowledge asserted
    disable data RAM
    negate Futurebus data strobe
**end if**
negate Futurebus address strobe
**wait until** falling clock edge
            and Futurebus address and data acknowledge both negated

negate Futurebus command
disable cache/Futurebus address and data transceivers

**if** ( not ( CPU request is write and required_transaction is read_shared
                                    and saved cache_status is set )
    and write_buffer is empty ) or abort_with_retry **then**
    cache_request <= false
    wait until middle of clock cycle and not cache_grant
**end if**

**if** abort_with_retry **then**
    return
**end if**

master_request_for_snoop_bus <= true after prop_delay;

determine current_cpu_attributes(replaced_section_index):
    set fb_valid
    **if** bus transaction was read_invalidate **then**
        set fb_owned and fb_exclusive
    **else**
        clear fb_owned
        **if** excl_depends_on_CS_on_read_shared
            and saved cache_status is set **then**
            set fb_exclusive
        **else**
            clear fb_exclusive
        **end if**
    **end if**
    clear fb_dirty

cpu_attributes(replaced_section_index)
    <= current_cpu_attributes(replaced_section_index)
lru_section <= new_lru_section
. . .
```

Figure 4-15 (continued).

```
        . . .
        enable cpu/snoop address transceiver in cpu–to–snoop direction

        snoop_attributes(replaced_section_index)
            <= current_cpu_attributes(replaced_section_index)

        **wait until** rising clock edge and master_granted_snoop_bus

        write enable replaced_section_index of cpu tag RAM
        write enable replaced_section_index of cpu attributes RAM
            (also writes lru section)
        write enable replaced_section_index of snoop tag RAM
        write enable replaced_section_index of snoop attributes RAM

        **wait until** falling clock edge

        master_requesting_snoop_bus <= false

        disable cpu tag RAM
        cpu_attributes(replaced_section_index) <= undriven
        lru_section <= undriven
        disable cpu attributes RAM
        disable cpu/snoop address transceiver
        disable snoop tag RAM
        snoop_attributes(replaced_section_index) <= undriven
        disable snoop attributes RAM

    **end procedure** sequence_read_for_miss
```

Figure 4-15 (continued).

ting up the Futurebus command values based on the required bus transaction type. It then sequences the Futurebus transaction, starting with an address beat. It saves the value of the Cache-Status reply signal for later use in updating the attributes of the line. In addition, it saves the state of the busy reply signal for use after the bus transaction is complete. If there is no busy reply, the transaction proceeds with a data beat, reading data into the data RAM. When the transaction sequence is complete, the procedure negates the Futurebus command values and disables the cache-to-Futurebus transceivers. Next, it determines whether to release Futurebus tenure. It does so only if the miss is not a write miss requiring a subsequent bus write transaction, and if the write buffer is empty, or if the read transaction returned busy status. If the Futurebus transaction returned retry status, no further action is required, so the procedure returns. Otherwise, the CPU and snoop tag and attributes for the fetched line must be stored. The new attribute values are determined, based on the transaction type and on the value of the configuration parameter excl_depends_on_CS_on_read_shared. The procedure arbitrates for access to the snoop bus, and sets up the new attribute values and LRU section index for the CPU and snoop attribute RAMs. The new tag value is already present on the cache address bus as part of the line's address. When access to the snoop bus is granted, the CPU and snoop tag and attribute RAMs are updated. The procedure then releases the snoop bus and disables all control signals used to update the tag and attribute RAMs.

Snoop Sequencer Outline

The snoop process shown in Figure 4-4 models the sequencing of cache datapath operations in response to Futurebus transactions. The process uses the clock signal generated by the processor to synchronize operations. A pseudo-code outline of the snoop process is shown in Figure 4-16. The process waits until a Future bus transaction is initiated, indicated by the Futurebus address strobe signal being asserted, then calls a procedure to sequence looking up the snoop attributes to check for a hit. The process next determines what action is required in response to the transaction. The decision is based on the P896.2 rules and the cache configuration parameters. If the lookup found a hit in the write buffer, the snoop remains unselected and will return a busy reply to the transaction master. If the lookup found a hit in the cache, the transaction is a broadcast and the parameter sel_on_broadcast_hit indicates that the snoop should accept the data, the snoop becomes selected. If the lookup found a hit, the line is owned by the cache and the transaction is not a broadcast, the snoop must become a third party in the transaction. If the transaction is a read-shared and the parameter re-

```
snoop : process is
begin
    futurebus_transaction_loop : loop
        wait until falling clock edge and Futurebus address strobe asserted

        sequence_snoop_lookup

        if write buffer hit then
             connection_state := unselected
        elsif fb_valid and broadcast and sel_on_broadcast_hit then
            connection_state := selected
        elsif fb_valid and fb_owned and not broadcast then
            if read-shared & reflect_on_read_shared then
                connection_state := reflecting
            else
                connection_state := intervening
            end if
        else
            connection_state := unselected
        end if

        if not write buffer hit then
            set new attributes to current attributes (initially)
            if cache_command then
                clear new fb_exclusive attribute
            end if
            if ownership or reflect then
                clear new fb_owned attributes
            end if
            if ( fb_valid and intent_to_modify
                    and ( (not broadcast and not fb_owned)
                            or (not broadcast and not write)
                            or (broadcast and unselected) ) )
                or third_party and inval_if_third_party then
                clear all new attributes
            end if
        end if
        . . .
```

Figure 4-16. Outline of the snoop process.

flect_on_read_shared is set, the snoop becomes a reflecting third party, otherwise it be-

```
        ...
        if write buffer hit then
            assert Futurebus busy reply
            negate Futurebus cache_status, selected, third_party and intervene reply
        else
            negate Futurebus busy reply
            if new fb_valid attribute set then
                assert Futurebus cache_status reply
            else
                negate Futurebus cache_status reply
            end if
            if connection_state is selected then
                assert Futurebus selected reply
                negate Futurebus third_party and intervene reply
            elsif connection_state is reflecting then
                assert Futurebus third_party reply
                negate Futurebus selected and intervene reply
            elsif connection_state is intervening then
                assert Futurebus third_party and intervene reply
                negate Futurebus selected reply
            else
                negate Futurebus selected, third_party and intervene reply
            end if
        end if
        assert Futurebus address acknowledge

        if write buffer hit or (unselected and new attributes same as old attributes) then
            snoop_requesting_line <= false
            snoop_using_snoop_bus <= false
            wait until falling clock edge and Futurebus address strobe negated
            negate Futurebus address acknowledge
        else
            sequence_snoop_participation
        end if

        disable snoop tag latch and set index latch outputs
    end loop futurebus_transaction_loop
end process snoop
```

Figure 4-16 (continued).

comes an intervening third party. In the remaining cases where the lookup found a hit, and in the case where the lookup found a miss, the snoop remains unselected.

Next, if there was no write buffer hit, the snoop process determines what attribute changes are required. The decision is based on the P896.2 rules. In addition, if the snoop is to participate as a third party and the parameter inval_if_third_party is set, the snoop must clear all attributes.

Having determined the required response and new attribute values for the line, the snoop process acknowledges the address beat. It asserts or negates the Futurebus reply signals as appropriate, and asserts the Futurebus address acknowledge signal. Next, the process implements its required action. If a write buffer hit was detected, or if the snoop is unselected and need not update the line's attributes, the snoop need take no further part in the transaction. It immediately releases the mutual exclusion lock on the line and use of the snoop bus, then waits for the end of the transaction. When the transaction completes, the process negates the address acknowledge. If the snoop does have some actions to perform, the process calls a procedure to sequence those actions. Upon completion of sequencing the transaction, the snoop process loops and waits for the next transaction.

The procedure that sequences lookup of the snoop attributes is shown in Figure 4-17. The procedure first requests arbitration for mutually exclusive access to the cache line, then speculatively enables the snoop attribute and tag RAMs. It then waits until mutual exclusion is granted and the master is not using the snoop bus before proceeding. The procedure confirms its own use of the snoop bus, thus locking out the master, and waits for the attribute values to be accessed and the tag comparison to be performed. During this time, the copyback address comparator also compares the Futurebus address with the saved address of the line awaiting copying back to shared memory. If there is a copyback pending and the addresses match, the procedure indicates a write buffer hit. Otherwise, it samples the current snoop attributes, and checks whether there is a hit in either section of the cache. Finally, the procedure disables the snoop tag RAM and attribute RAM outputs and returns.

```
procedure sequence_snoop_lookup is
begin
    snoop_requesting_line <= true

    enable snoop tag latch and set index latch outputs
    enable snoop tag ram comparison
    enable snoop attributes ram outputs

    wait until rising clock edge
              and master not using snoop bus and snoop granted mutex to line

    snoop_using_snoop_bus <= true
    wait until falling clock edge

    if copyback_pending and copyback comparator match then
        set write buffer hit
    else
        sample current snoop attributes
        assume initially hit is false
        for each section loop
            if tag match and cpu attribute fb_valid is set then
                hit is true, hit_section_index is this section
            end if
        end loop
    end if

    disable snoop tag ram comparison
    disable snoop attributes ram outputs
end procedure sequence_snoop_lookup
```

Figure 4-17. Outline of the procedure that sequences snoop attribute lookup.

The procedure that sequences the snoops actions during a transaction is shown in Figure 4-18. The procedure first requests arbitration for use of the cache buses and speculatively enables the address transceivers to pass the Futurebus transaction address to the cache address bus. It then waits for use of the cache buses to be granted. If the snoop must participate in the transaction, the procedure then enables the data transceivers between the cache data bus and the Futurebus and enables the data RAM. The direction of data transfer depends on whether the transaction is a read or a write. The procedure then waits until either the data strobe is asserted, indicating commencement

```
procedure sequence_snoop_participation is
begin
    snoop_request_for_cache_bus <= true
    enable cpu/snoop address transceiver in snoop_to_cpu direction

    wait until falling clock edge and snoop_granted_cache_bus

    if not unselected then
        if Futurebus write then
            enable cache/Futurebus data transceivers
                in Futurebus-to-cache direction
            write enable hit section of data RAM
        else
            enable cache/Futurebus data transceivers
                in cache-to-Futurebus direction
            output enable hit section of data RAM
        end if

        wait until falling clock edge
                    and Futurebus data strobe asserted or address strobe negated

        if Futurebus data strobe asserted then
            assert Futurebus data acknowledge
            if Futurebus write then
                disable write to data RAM
            end if
            wait until falling clock edge
                        and Futurebus data strobe negated and address strobe negated
            if not Futurebus write then
                disable data RAM output
            end if
            negate Futurebus data acknowledge
        end if

        enable cache/Futurebus data transceivers

    else
        wait until falling clock edge and Futurebus address strobe negated
    end if

    if new attributes differ from current attributes then
        enable cpu attribute ram outputs

        wait until rising clock edge
        sample current_cpu_attributes and current_lru section
        disable cpu attribute ram outputs
        . . .
```

Figure 4-18. Outline of the procedure that sequences the snoop's actions in a transaction.

```
        . . .
        update current_cpu_attributes(hit_section_index) with new attributes
        if new fb_owned is clear then
            clear current_cpu_attributes(hit_section_index).dirty
        end if
        cpu_attributes(hit_section_index)
            <= current_cpu_attributes(hit_section_index)
        lru_section <= current_lru section
        snoop_attributes(hit_section_index) <= new attributes
        write enable hit_section_index of cpu attributes ram
            (also writes back lru section)
        write enable hit_section_index of snoop attributes ram

        wait until falling clock edge
        cpu_attributes(hit_section_index) <= undriven
        lru_section <= undriven
        disable cpu attributes ram
        snoop_attributes(hit_section_index) <= undriven
        disable snoop attributes ram
    end if

    disable cpu/snoop address transceiver
    negate Futurebus address acknowledge

    snoop_requesting_line <= false
    snoop_using_snoop_bus <= false
    snoop_request_for_cache_bus <= false

end procedure sequence_snoop_participation
```

Figure 4-18 (continued).

of a data beat, or until the address strobe is negated, indicating and address-only transaction. In the case of a data beat, the procedure asserts the data acknowledge signal, since, by this time, data has been accepted for a write or data is available for a read. For a write, the procedure also disables the data RAM to complete acceptance of the data. The procedure then waits until completion of the data beat and commencement of the end beat, indicated by both the data strobe and the address strobe being negated. For a read, it disables the data RAM output, since the transaction master has accepted the data. The procedure negates the data acknowledge signal to acknowledge completion of the data beat and disables the data transceivers. In the case of the snoop not participating in the transaction, the procedure ignores any data beat that occurs, and just waits for the end beat, indicated by the address strobe being negated.

The snoop's action during the end beat is to update the stored CPU and snoop attributes if the new attribute values differ from the current values. The procedure enables the CPU attribute RAM output in order to read the additional attributes not replicated in the snoop attribute RAM. It then waits for the values to be read, samples them, and disables the CPU attribute RAM outputs. It updates the attribute values for the hit section, and clears the Dirty attribute if ownership is relinquished. The procedure then drives the updated attributes onto the CPU and snoop attribute buses, and enables the CPU and snoop attribute RAMs to write the new values. After a delay for the write to occur, it removes the driving values and disables the attribute RAMs. The final actions of the procedure are to disable the address transceiver between the snoop and cache buses, to acknowledge completion of the end beat by negating the address strobe, to release the mutual exclusion lock and to release use of both cache and snoop buses.

Write Buffer Sequencer Outline

The process that sequences write transactions from the write buffer is shown in Figure 4-19. All action are synchronized with the falling clock edge in the middle of a clock cycle. The process waits until there is an entry waiting at the output of the write buffer FIFO. It then checks whether it already is granted use of the Futurebus or whether the master process is waiting for the write buffer to be flushed. If neither is the case, the process requests arbitration for the Futurebus, and waits until access is granted. When the process can proceed, it prepares for the Futurebus transaction by setting up the Futurebus command values and by enabling address and data outputs from the write buffer FIFO. The Cache-Command value is set using the cache_com-

```
write_buffer_controller : process is
begin
    wait until falling clock edge and write buffer not empty

    if not (write_buffer_grant or flush_write_buffer) then
        write_buffer_request <= true
        wait until falling clock edge
                    and (write_buffer_grant or flush_write_buffer)
    end if

    assert/negate Futurebus cache_command
        depending on cache_command mark output
    assert/negate Futurebus intent_to_modify
        depending on write_through mark output
    assert Futurebus write
    negate Futurebus broadcast, three_party and ownership

    enable write buffer address and data outputs

    assert Futurebus address strobe
    wait until falling clock edge and Futurebus address acknowledge asserted
    save Futurebus busy reply value
    if not Futurebus busy reply then
        assert/negate Futurebus three_party depending on third_party reply
        assert Futurebus data strobe
        wait until falling clock edge and Futurebus data acknowledge asserted
        enable write buffer shift-out
        negate Futurebus data strobe and address strobe
        if write_through mark output is cleared then
            reset copyback_pending flip-flop
            reset copyback address comparator
            wait until rising clock edge at start of next clock cycle
            disable write buffer shift-out
        else
            wait until rising clock edge at start of next clock cycle
            disable write buffer shift-out
        end if
        wait until falling clock edge
                    and Futurebus address and data acknowledge both negated
    else
        negate Futurebus address strobe
        wait until falling clock edge
                    and Futurebus address acknowledge negated
    end if
    . . .
```

Figure 4-19. Outline of the process that sequences write transactions from the write buffer.

```
      . . .
      negate Futurebus command
      disable write buffer address and data outputs
      if write_buffer_preempt  or write buffer empty
          or saved Futurebus busy reply is set then
          write_buffer_request <= false
      end if
  end process write_buffer_controller
```

Figure 4-19 (continued).

mand mark output from the FIFO. The Intent-to-Modify value is set using the write_through mark: set for a write-through or cleared for a copy-back. The process then sequences the Futurebus transaction, starting with an address beat. It saves the state of the busy reply signal for use after the bus transaction is complete. If there is no busy reply, the transaction proceeds with a data beat, writing data from the FIFO output. On completion of the data beat, the process shifts the entry out of the head of the FIFO and terminates the Futurebus transaction. In the case of a copy-back, the process also resets the copyback pending flip-flop to indicate to the master process that the copy-back has completed, and resets the copyback address comparator. By the time the entire Futurebus transaction is completed, the next entry in the FIFO (if there is one) has advanced to the head, and the master has determined whether it needs to maintain the flush_write_buffer signal active.

If there was a busy reply status from the address beat of the bus transaction, the process does not proceed with the data beat. Instead, it terminates the bus transaction and leaves the entry at the head of the write buffer FIFO so that the write can be retried later.

When the transaction sequence is complete, with or without retry status, the process negates the Futurebus command values and disables the address and data outputs from the write buffer FIFO. It then determines whether to release Futurebus tenure. It does so if it is preempted by the master, if the FIFO is empty, or if the write transaction returned retry status. The process then repeats from the beginning.

4.3.4 The Coherence Monitor Model

The coherence monitor within the Leopard-2 system model consists of a set of observation processes, one within each cache, and a global monitor that verifies maintenance of coherence. The observation process in a cache senses the values written to and read from the cache and senses the changes made to attribute values of lines in the cache. It communicates this information to the global monitor.

In order for the global monitor to verify maintenance of coherence for a line in the shared memory address space, it must verify two properties:

- that the attributes of the line are maintained within the legal set of configurations for the coherence protocol being used, and
- that each cache provides its client processor with the most recently written version of the line.

The global monitor checks the first property by maintaining, for each line, a copy of the attributes stored in the caches. This represents the configuration of the line. The monitor has a generic parameter that specifies the coherence protocol in use and thus which configurations are legal. When an observation process informs the global monitor of a change in attribute values for a line in a cache, the monitor updates its copy and verifies that the new configuration is legal.

The global monitor checks the second property by maintaining a copy of the *shared memory image*, which is the view of shared memory that should be seen by each processor. The shared memory image includes updates from processor write operations, even if they are not transmitted through to the physical shared memory. Recall that the Leopard-2 model represents the value of a line by a token stamped with the time at which the line was last updated by a processor write. If the coherence protocol is correct, each cached copy of a line should contain the same latest updated value. On a read hit in a cache, the observation process informs the global monitor of the timestamp of the value read from the cache. The monitor compares the timestamp with that of the shared memory image line to ensure that they are equal. Any write to a cache updates the shared memory image, so the observation process informs the global monitor. The monitor compares the timestamp of the update with that of the shared memory image to ensure that the superseded value is older. If it is not, the protocol or its implementation is in error.

One difficulty that arises in implementing the global monitor is that of the size of the shared memory image. There is potential for the host simulator to run out of memory. The same problem arises in implementing behavioural simulation models of large memory systems, and is solved by using sparse representations of the address space, storing only those sections that are actually in use. In the case of the global monitor model, the address space is divided into segments, each the size of one cache section. The first time that any line within a segment is cached, storage is allocated for that segment of the shared memory image. Any segment without allocated storage is deemed to have all lines uncached (invalid in every cache) with a timestamp of 0.

The observation process in each cache, as mentioned above, informs the global monitor of read and write values and attribute changes. The process senses the state of control signals generated by the cache controller to determine when to sample data and attribute values. A read value is sampled when the cache controller enables the cache data RAM output to service a read hit. A write value is sampled when the cache controller write-enables the data RAM to service a write hit. In the case of a write hit to a shared line involving a Futurebus transaction, the transaction is performed before the cache is updated. If the transaction is a write-update, another cache may update its value and read it before the write to data RAM occurs in the writing cache. To avoid this being detected as an error by the global monitor, the observation process in the writing cache also samples the write data when the cache controller initiates a Futurebus write transaction for a write hit to a shared line.

The observation process samples attribute values whenever the master or snoop write-enables the CPU attributes RAM, and informs the global monitor of the new value. It uses the tag and set index values from the cache address bus to determine the address of the line for use by the monitor.

A complication arises when a line is replaced by the master. The global monitor must be informed so that it can modify the the attributes for the line to indicate that the line is no longer valid in that cache. The difficulty is that, if the line is not dirty, the entry is simply overwritten in the cache without the tag of the replaced line being output onto any bus. Hence the observation process is unable to determine the address of the replaced line just by sampling the cache buses. To circumvent this problem, the observation process maintains its own copy of the cache tags. When the cache controller write-enables the CPU tag RAM, the observation process informs the global monitor that a line is being replaced. The address of the line is determined using the set index value on the cache address bus and the tag value saved by the observation process. The observation process then copies the new tag value.

4.4 Summary

This chapter has described the design of a programmable cache controller for the Leopard-2 Multiprocessor. The programmability is based on an analysis of the options within the P896.2 Futurebus cache coherence rules, and allows implementation of the various cache coherence strategies discussed in Chapter 2. This chapter also describes a behavioural model of the Leopard-2 system, and details the control sequences used by the cache controller to manage the cache datapath. The model, in execution, is driven by two alternative workloads: a synthetic workload to exercise specific aspects of the system behaviour, and a pseudo-random workload to provide comprehensive test coverage. The model also incorporates monitors to verify correct maintenance of coherence by the caches.

Chapter 5
Conclusions

5.1 Summary of Project Context

In Chapter 2 of this thesis, three approaches to evaluating performance of cache coherence strategies were compared: analytical, simulation based, and by measuring real systems. The third approach yields the most accurate evaluation, provided differences due to other factors can be isolated.

The author's part in the Leopard Project was to design and construct an experimental vehicle for making such measurements. This sub-project formed the basis for further work in the overall project, including:

- using the experimental hardware to evaluate the protocols described in detail in Chapter 2 under real operating conditions,
- experimental work in multiprocessor operating systems,
- development and evaluation of concurrent applications, including the Multiview software engineering environment [1] and a parallel implementation of the SISAL functional language [22],

The motivation for constructing a system to allow measurements of cache behaviour came from the scarcity of raw data available, and the lack of validated comparisons published in the literature. This was despite the frequently cited advantages of the shared memory multiprocessor architecture, and the prediction that future workstations and network servers would rely heavily on that architecture. At the time of commencement of the project, none of the small number of commercially available multiprocessors had the flexibility to allow the experimental goals to be met.

One approach considered as a means of developing a system for cache coherence experiments was to accept some of the limitations inherent in existing commercial systems, and to modify them sufficiently to support different cache coherence strategies. This would have involved less design and construction work than designing a new system, since the existing input/output and memory systems would be used. However, it would have relied on a commercial system having a sufficiently flexible bus protocol to allow augmentation (possibly including extra bus wires) to implement different cache coherence protocols, and having hardware design and packaging organized in such a way as to allow "splicing in" extra cache hardware. Furthermore, it would have relied on a vendor being willing to make available the proprietary design information to support such modifications to their hardware. Discussions held with some vendors and other researchers in the early stages of this project indicated that vendors regard such proprietary information as extremely confidential and sensitive, and are not willing to make it available to independent researchers, even under strict secrecy agreements. Hence, for these reasons, if not for reasons of technical difficulty, this approach was rejected.

With this background as a starting point, design and construction work on the Leopard systems was undertaken. The project was extremely ambitious, but construction of a complete Leopard-2 system has been completed and the system demonstrated. The prototype consists of four General Data Processors, a Shared Memory, a Futurebus Monitor and a Storage and Communications Processor. Subsequent work included porting the Chorus operating system [43] to run on the hardware.

5.2 Experimental Evaluation of Cache Coherence Protocols

Of the experiments that the Leopard multiprocessor was designed to support, the one most relevant to this thesis is the measurement of system performance under different cache coherence protocols. It is essential to realize that a complete computer system consists not just of the CPU, cache and main memory, but also the I/O hardware, the operating system software, and the applications programs that the end user runs. Different cache coherence protocols may affect the overall system performance to different degrees as these other factors are included. Thus a valid performance evaluation experiment must include the effects of these additional factors, and, in reporting results, must also report the conditions under which the measurements were made.

One factor of great interest is the way in which performance under different protocols is affected by the choice of workload for the system. The way in which the protocols deal with data shared between processors is one of the primary differences between them, so it is important to assess performance under workloads with differing degrees of data sharing. Some studies, for example those done by Eggers and Katz [18], suggest that the amount of actual data sharing in real applications is low. However, the extent to which the applications they measured are typical of parallel programs that will be run on shared memory multiprocessors is questionable. One could argue that if sharing is efficiently supported by the architecture, software designers will write programs in such a way as to take advantage of it, thus increasing the degree of sharing above that measured in current programs.

At the time of construction of the Leopard multiprocessor, the experiments planned involved using a variety of application programs to run as benchmarks. The chosen programs spanned a spectrum of process granularity and data sharing:

- the parallel SISAL system, representing fine grained parallelism with heavy data sharing,

- the Multiview system, representing coarse grained parallelism with a lower degree of data sharing,
- data parallel algorithms, such as image processing and numerical applications,
- general interactive workstation usage, such as office automation and software development tasks.

These benchmark applications were to run under the Chorus micro-kernel operating system [43] ported to the Leopard-2 system, supplemented with a Unix interface layer. The planned cache evaluation experiments involved running each benchmark with each of the cache coherence protocols, and forming a matrix of performance measurements. The primary metric of performance would be execution time of the program being run. It is well recognized that this is the most reliable and complete measure of system performance, as it includes all of the effects of hardware organization, I/O operation, operating system overhead and application characteristics. (See, for example, [26] for a discussion of this issue.)

One of the difficulties with this experimental technique is maintaining a controlled environment, so that only the desired factors vary between runs. Two techniques were considered to deal with this problem. The first was to use a fixed initial configuration for each trial, and to isolate the system from random external influences. Setting up an initial configuration involves such factors as formatting a disk store with exactly the same files and data in the same locations, and starting the trial by boot-strapping from this initial configuration. Isolation from external influences involves disconnecting any network connections to other systems, and using scripts to control the running of the benchmarks, so as not to rely on human user reaction times. The second technique for dealing with variations in the environment was to perform a number of trials of each measurement, and average the results. While this has the advantage of including the effects of external influences on the computer system performance, the additional time required to complete sufficient trials to gain statistical significance may make the experiment intractable.

5.3 Conclusion

This thesis describes research in the area of cache coherence protocols for bus connected shared memory multiprocessors, undertaken as part of the Leopard Project. The research has produced some significant theoretical results, which underpin a practical experiment in the construction of a multiprocessor platform for evaluating cache coherence protocols.

The description of cache coherence protocols in a uniform framework, presented in Chapter 2, allows comparisons to be made between the protocols, and simplifies their analysis. This work, and the research into bus protocols performed whilst designing the Leopard-1, had a significant impact on the design of cache coherence mechanisms for the IEEE Futurebus Standard. These, in turn, made possible the design of the general purpose cache datapath in the Leopard-2 system (described in Chapter 3) and the reconfigurable cache controller (described in Chapter 4), which can be programmed to implement different coherence protocols.

The Leopard-2 multiprocessor forms a vehicle for comparing the performance of a variety of cache coherence protocols under controlled conditions. This form of evaluation is necessary to validate previous performance studies done using analytical and simulation based techniques. The Leopard-2 system is now operational, and may be used to perform such experiments. The results of these experiments will greatly aid future designers of shared memory multiprocessors to achieve maximum performance for programs run on this important class of computer systems.

Appendix A
L-Bus Data Transfer Protocol

This appendix describes the data transfer protocol developed for the Leopard-1 Multiprocessor system.

A.1 Overview

L-Bus transactions are composed of an address transfer from a master to slave modules, followed by zero or more data transfers between them. When a module needs to initiate bus transactions, it requests access to the L-bus using the arbitration protocol. When it is granted the bus, it becomes master, and may initiate zero or more transactions. (A typical case where no transactions are done is when the master must perform an interlocked data access, but some other module already owns the lock. This is described in detail below.) When the master has completed its last transaction, it releases the bus.

During an address transfer, the master broadcasts the first address for the transaction. Each module determines whether it will be involved in the transaction, and if so, takes a copy of the address, and increments it after each data transfer. The particular module referred to by the address is selected as the slave. Other modules may be involved as third parties, and may monitor the subsequent data transfers.

For each data transfer, the caches determine whether they have a copy of the data being accessed. If a cache does not have a copy, it does not participate further in the data transfer. If a cache does have a copy, then its response depends on the type of data transfer, as discussed below

The L-Bus data transfer protocol supports both write-through and write-back caches, and provides mechanisms to support cache coherence protocols. Caches may intervene on data transfers to supply the most up-to-date copy of a block, and may notify other caches that they should invalidate their copies of a block. The L-Bus protocol implements these mechanisms in a way that allows a mixture of cache coherence protocols in the one system. However, it assumes that all caches use the same block size.

The L-Bus data transfer protocol also implements an interlocking mechanism for controlled access to shared data. A lock signal is provided, which only one module may assert at a time. When a module needs exclusive access to data, it requests the bus and checks the state of the lock signal. If the signal is asserted, the module must relinquish the bus and retry later. If the signal is negated, the module asserts it and commences interlocked operation on the data. The operation may continue over more than one bus tenure, and non-interlocked data access by other modules may be interleaved. During the last data transfer of the interlocked operation, the module negates the lock signal.

A.2 Addressing Structure

Addressing in L-Bus is based on backplane slot position. Slots are numbered from 1 to 30 from left to right (viewed from the board insertion side) across the backplane. The slot number is encoded on signals presented to each board by the backplane. Each board is allocated a region in the address space according to its slot number.

The address space of a board is divided into two regions, one for cachable data and one for non-cachable data. Modules accessing data from the cachable region may put the data in their local caches. All caches must monitor data transfers to and from this region. Data from the non-cachable region must not be cached.

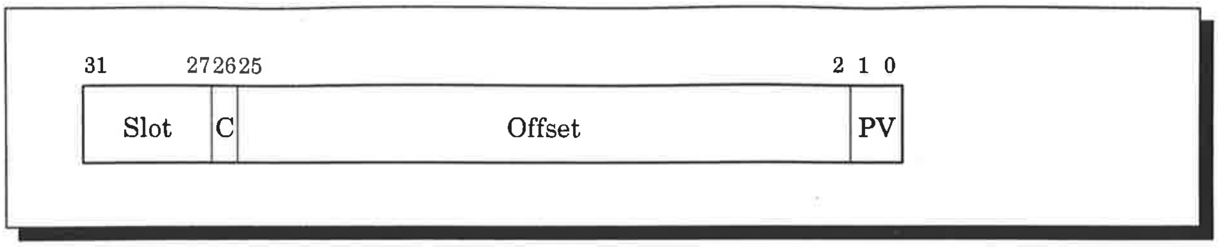


Figure A-1. L-Bus address format.

The format of an L-Bus address is shown in Figure A-1. The Slot field indicates which backplane slot contains the board to be selected as the slave. Slot address 0 is reserved for modules to access local memory and registers. Slot address 31 is reserved for the System Region (see below). When the C field is 1, the offset is in the cachable region of the board's address space, otherwise it is in the non-cachable region. The Offset field is the word address of the data within the selected region of the board. The byte address is four times the word address. The PV field is the protocol version indicator. It indicates the lowest version of the L-Bus protocol which implements the type of transaction requested by the master. If the slave conforms only to some lower version and cannot perform the requested transaction, it must abort the transaction. The use of the protocol version field allows enhancements to be made to the bus protocols whilst ensuring compatibility with existing modules. The restrictions applying to to different protocol versions are described in Section A.4.13 below. The values of PV defined are 00 for the Quibus Version 0.9 bus protocol (used in the QDS-1000 system), and 01 for the L-Bus protocol described here.

The System Region of the L-Bus address space, corresponding to Slot address 31, is reserved for registers and storage not specific to a particular module, but which are used to coordinate modules in a system. Locations in the System Region may have a distributed implementation, or be decoded in a way dependent on the state of the system. One such location is the Interrupt Identifier location, at offset 0 in System Region. This is used in the interrupt protocol to acknowledge receipt of an interrupt. The actual module that responds to an access to this location is the one selected by the interrupt mechanism as the winning requester.

A.3 Data Transfer Signals

The L-Bus specification defines a number of backplane signals to carry address, data, command and status information, and to synchronize transfer of this information between modules. These signals are described in this section.

A.3.1 Information Signals

Inf[0:31] — address and data values

Address values are formatted as described in Section A.2, with bit 0 on Inf[0]. Data values are four bytes wide, with the lowest address (byte 0) on Inf[0:7], the next byte on Inf[8:15], etc.

A.3.2 Master Command Signals

Command[0:3] — carry command information during an address transfer and byte-enable information during a data transfer

EndSeq — last data transfer in a transaction

Lock — interlocked operation in progress

The command information sent during an address transfer specifies the type of the transaction, and hence whether the subsequent data transfers are reads or writes. The command values are:

Command				Transfer Type
3	2	1	0	
0	0	0	0	non-cache read
0	0	0	1	non-cache write
0	0	1	x	*reserved*

0	1	x	x	*reserved*
1	0	0	0	cache read-shared
1	0	0	1	cache write-back
1	0	1	0	cache read-invalidate
1	0	1	1	cache write-invalidate
1	1	0	0	cache read-copy
1	1	0	1	cache write-copy
1	1	1	0	*reserved*
1	1	1	1	cache immed-invalidate

During a data transfer the command signals are used to send byte-enable information to the slave. Command[0] is used as ByteEn[0] controlling byte 0, Command[1] as ByteEn[1] controlling byte 1, etc. In a write transfer, only those bytes with the corresponding ByteEn signal asserted are written, the other bytes in the word being preserved. In a read transfer, only those bytes with the corresponding ByteEn signal asserted are required by the master.

A.3.3 Cache Status Signals

PassiveHit — a cache has a copy of the data, and it is consistent with shared memory

InterveneHit — a cache has the most up-to-date copy of the data and must intervene on the transfer

SlaveUpdate — the memory slave must update its copy of the data with the value transferred on the bus

A.3.4 Slave Status Signals

Status[0:2] — slave completion status for the transfer

The values returned for completion status are as follows:

Status			Code	Indicated Condition
2	1	0		
0	0	0	SelectErr	Slave does not respond
0	0	1	AccErr	Access error

0	1	0	OpErr	Invalid operation requested
0	1	1	ProtErr	Protocol error
1	0	0	Succes	Successful completion
1	0	1	Retry	Busy retry
1	1	0		*reserved*
1	1	1	TPAbort	Third party abort

The code Success indicates that no problem occurred during the data transfer. Retry is returned when the slave is busy, and waiting for it would unduly hold up the bus or cause deadlock. When this status is returned, the master must terminate the current transaction, release the bus, request it again, and retry the data transfer. SelectErr is returned when there is no board inserted into the addressed slot, or when the board's address space does not include the addressed offset. AccErr is returned when the slave is unable to complete the data transfer because of an internal error, such as a memory checksum error or a component fault. OpErr is returned when the transfer required an invalid operation to be performed, such as writing to read-only memory or reading a word from a byte-wide register. ProtErr is returned when the protocol level of the data transfer is higher than the maximum recognized by the slave. TPAbort is returned when a third party module needs to abort the transfer.

A.3.5 Sequencing Signals

AddrStrobe — master has initiated an address transfer

DataStrobe — master has initiated a data transfer

Ready — all modules are ready for data transfer (wired-and)

CacheAck — all caches acknowledge availability of cache status (wired-and)

StatusAck — all participating modules acknowledge availability of status (wired-and)

TPAck — all third party modules acknowledge completion of data transfer (wired-and)

A.3.6 Slot Address

Slot[0:4] — slot number, supplied by the backplane

A.4 Data Transfer Protocol Operation

In this section, the operation of the L-Bus data transfer protocol is described in detail. Timing diagrams for each of the transfer types are shown in Section A.5.

A.4.1 Information Transfer Handshaking

The transfer of information (address and data) on the Inf signals is sequenced with a handshaking protocol using the synchronization signals listed in Section A.3.5 above. This section firstly describes the basic handshaking protocol used for non-caching transactions, then describes how it is augmented for transactions involving cachable data.

Initially, when the data transfer bus is idle, each module negates AddrStrobe, DataStrobe, StatusAck and TPAck, and asserts Ready when it is ready to commence a transfer. Since Ready is wired-and, it will not become true until all modules are ready.

When a master must initiate a transfer, it waits until Ready is true, then places a command code on the Command signals, and may place information on the Inf signals. If the transfer is an address transfer, the master asserts AddrStrobe, whereas if the transfer is a data transfer, the master asserts DataStrobe.

When one of the strobe signals becomes true on the bus, each module sequences the Ready, StatusAck and TPAck signals according to the part they must play in the transfer protocol. The master module itself immediately negates Ready and asserts StatusAck and TPAck. The addressed slave module and the third party modules negate Ready and commence their action for the transfer.

When the slave has completed its action, which may involve placing information on the Inf signals, it places a status value on the Status signals and asserts StatusAck.

Any third party module which needs to signal an error may do so by forcing the Status value to TPAbort and asserting StatusAck. Other third party modules also assert StatusAck. Since this signal is wired-and, it will not become true until all participating modules have determined the status for the transfer.

When each third party module has completed its action, it asserts TPAck. Since this signal is wired-and, it will not become true until all third party modules have completed their actions.

When the master sees StatusAck and TPAck both true, it removes the command and any information from the bus and negates the strobe signal, StatusAck and TPAck. All other modules, on seeing the strobe signal false, remove any status or information from the bus and also negate StatusAck and TPAck. All modules hold Ready false until they are ready to commence the next transfer. When all modules assert Ready, the next transfer may commence.

In order to handle transfer of cachable data, the basic protocol is extended using the cache status signals. As in the basic protocol, the master waits until the Ready signal is true, places a command and possible data on the bus, then asserts DataStrobe. In a cachable data transfer, each cache then negates Ready and checks for a hit at the addressed location. If a cache does not have a hit, it does not assert any of the cache status signals. If a cache has a copy of the data which is consistent with the copy in shared memory, is asserts PassiveHit. If there is a cache which has a more recently written copy of the data than shared memory and hence must intervene on the data transfer, it asserts InterveneHit. It may also assert SlaveUpdate if it requires that the shared memory slave update its copy using the data transferred. When each cache has completed the check, it asserts CacheAck and StatusAck. Since CacheAck is wired-and, it does not become true until all caches have completed the check.

In a cachable data transfer, the slave waits until both DataStrobe and CacheAck are true, and then checks the cache status signals. If InterveneHit is false, it continues normally. If InterveneHit is true and SlaveUpdate is false, it suppresses its action and allows the cache asserting InterveneHit to complete the transfer. If InterveneHit and SlaveUpdate are both true, the slave allows the intervening cache to complete the transfer, but takes a copy of the transferred data to update its own copy. The remainder of the handshaking sequence then continues as in the basic protocol, with the caches removing the cache status signals and negating CacheAck when they see DataStrobe become false.

A.4.2 Address Transfer and Incrementing

A module which needs to use the data transfer bus firstly goes through the arbitration process, and when granted the bus, becomes the new bus master and initiates transactions.

To start a transaction, the master first initiates an address transfer, using the basic handshaking protocol described in Section A.4.1 above. The command code it sends on the Command signals indicates the type of the ensuing data transfers, and the information on the Inf signals is the address of the first data transfer. If there is a module in the addressed slot, it stores the address and command, and becomes the slave for the transaction. It returns an appropriate status code depending on whether or not it can continue with the transaction. Any third party modules may also store the address and command, and may return the TPAbort code if they need to prevent the transaction from proceeding.

After the address transfer has successfully completed, the master may initiate zero or more data transfers. The address for each data transfer is that stored by all of the participating modules, and must be incremented by one word after each successful data transfer in the transaction. When the master initiates the last data transfer in the transaction, it asserts EndSeq. In the case of a transaction with no data transfers, EndSeq is asserted during the address transfer. If there is a data transfer in which the status code returned is not Success, the transaction is terminated.

A.4.3 Cache Immed-Invalidate Transaction

A cache immed-invalidate transaction may only occur in the cachable region of a module's address space. If the master issues a cache immed-invalidate command code in an address transfer, the transaction contains no subsequent data transfers. The command indicates that any cache which has a copy of the data at the addressed location must immediately invalidate that copy.

A.4.4 Cache Read-Shared Transaction

A cache read-shared transaction may only occur in the cachable region of a module's address space. It is used by a cache master to read a block without changing the data in other caches. A third party cache which does not have a hit or which has a passive hit at the addressed location takes no action. A third party cache with an intervene hit must intervene to supply the data.

A.4.5 Cache Write-Back Transaction

A cache write-back transaction may only occur in the cachable region of a module's address space. It is used by a cache master which has the most recently written copy of the addressed block to write it back to memory. A third party cache which does not have a hit or which has a passive hit at the addressed location takes no action. If there is a third party cache with an intervene hit, the cache coherency protocol has been corrupted, so the cache should return the TPAbort status code.

A.4.6 Cache Read-Invalidate Transaction

A cache read-invalidate transaction may only occur in the cachable region of a module's address space. It is used by a cache master to gain an exclusive copy of the addressed data. A third party cache which does not have a hit at the addressed location takes no action. A third party cache with a passive hit must invalidate its copy of the data. A third party cache with an intervene hit must intervene to supply the data, and then invalidate its copy.

A.4.7 Cache Write-Invalidate Transaction

A cache write-invalidate transaction may only occur in the cachable region of a module's address space. It is used by a cache master to write its copy of the addressed data through to shared memory. A third party cache which does not have a hit at the addressed location takes no action. A third party cache with a passive hit must invalidate its copy of the data. A third party cache with an intervene hit must invalidate its copy and force shared memory to be updated.

A.4.8 Cache Read-Copy Transaction

A cache read-copy transaction may only occur in the cachable region of a module's address space. It is used by a cache master to gain a non-exclusive copy of the addressed data. A third party cache which does not have a hit or which has a passive hit at the addressed location may take a copy of the transferred data. A third party cache with an intervene hit must intervene to supply the data.

A.4.9 Cache Write-Copy Transaction

A cache write-copy transaction may only occur in the cachable region of a module's address space. It is used by a cache master to broadcast a copy of the addressed data through to shared memory and to third party caches. A third party cache which does not have a hit at the addressed location may take a copy of the transferred data. A third party cache with a passive hit must take a copy. A third party cache with an intervene hit must intervene to accept the data, and may force shared memory to be updated.

A.4.10 Non-Cache Read Transaction

A non-cache read transaction may occur at any address, and is the normal transaction type used by a non-cache master. If the addressed location is in the non-cachable region of a module's address space, the basic handshaking protocol is used. If the location is in the cachable region, the extended handshaking protocol is used. A third party cache which does not have a hit or which has a passive hit at the addressed location takes no action. A third party cache with an intervene hit must intervene to supply the data.

A.4.11 Non-Cache Write Transaction

A non-cache write transaction may occur at any address, and is the normal transaction type used by a non-cache master. If the addressed location is in the non-cachable region of a module's address space, the basic handshaking protocol is used. If the location is in the cachable region, the extended handshaking protocol is used. A third party cache which does not have a hit at the addressed location takes no action. A third party cache with a passive hit must invalidate its copy of the data. A third party cache with an intervene hit must intervene to accept the data, and may force shared memory to be updated.

A.4.12 Interlocked Transactions

Modules needing to perform interlocked operations on data use the Lock signal to ensure mutual exclusion. Before commencing an interlocked operation, a module checks this signal. If it is already true, some other module is in the middle of an interlocked operation. Hence the bus must be relinquished and requested again.

When the module has the bus and the Lock signal is false, the interlocked operation may commence. The Lock signal must be asserted before the first interlocked transfer, and held until the last data transfer in the interlocked operation. The bus may be relinquished in the midst of an interlocked operation to allow non-interlocked transactions to proceed. The Lock signal must be asserted for all transfers in an interlocked operation, not just while the bus is relinquished during the operation. This ensures correct interlocking in multi-port slaves, such as a memory block on an intelligent device controller.

A.4.13 Protocol Version

As indicated in Section A.2, protocol version 01 is used to indicate conformance with the data transfer protocol described here. Protocol version 00 is used to provide compatibility with the Quibus Version 0.9 bus protocol used in the QDS-1000 system. The restrictions applied for compatibility are:

- all data transfers must be of the non-cache read or non-cache write type, and
- each transaction must consist of an address transfer followed by at most one data transfer.

Where an L-Bus transaction could be performed according to these restrictions, it is done as a protocol version 00 transaction. This ensures maximum compatibility between modules, allowing QDS-1000 modules to be integrated into a Leopard-1 system.

A.5 Timing Diagrams

The following timing diagrams show the sequencing of address and data transfers in the L-Bus data transfer protocol. Each kind of transfer is shown from the perspective of the master, the slave and a third party module.

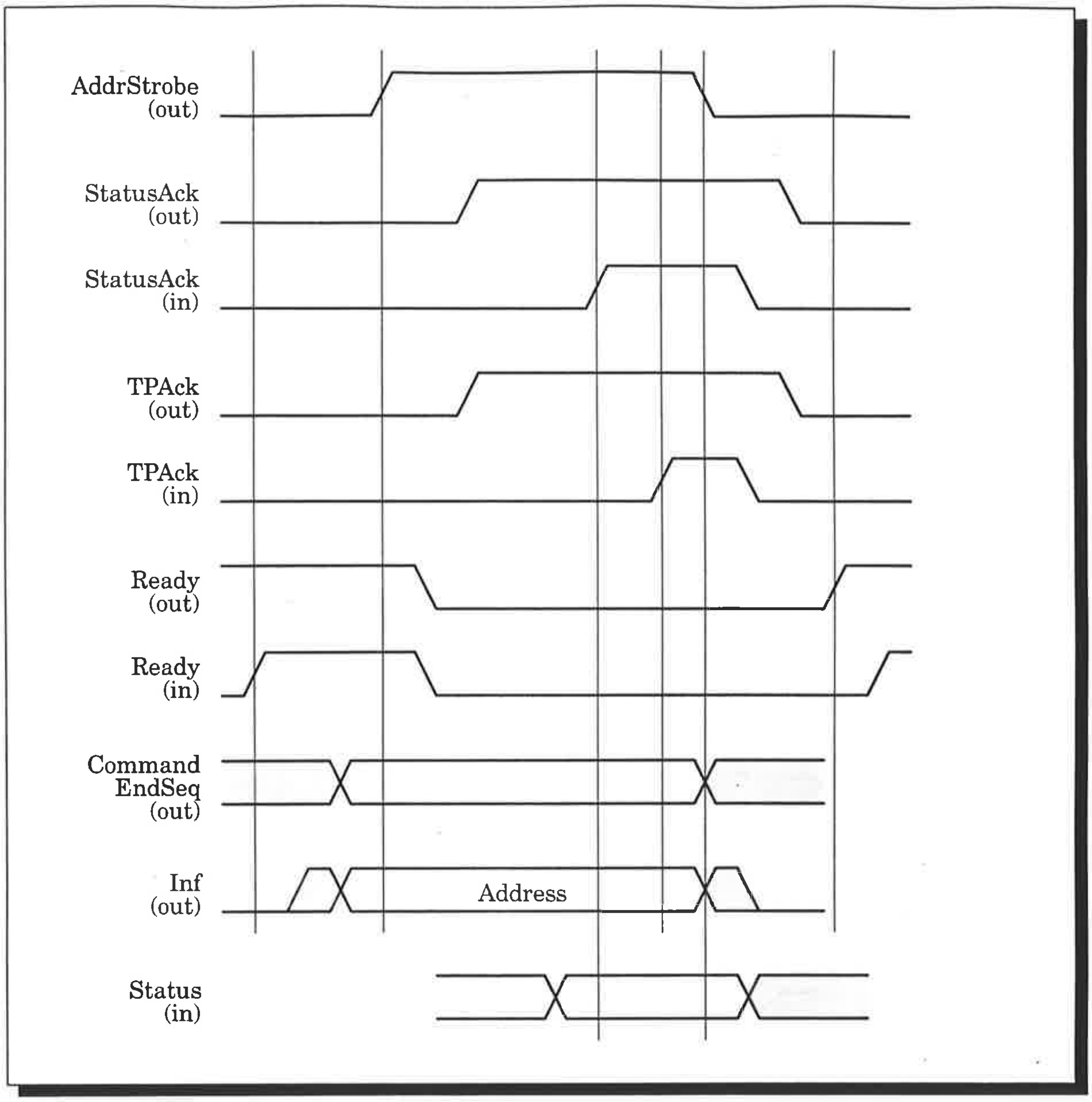


Figure A-2. Address transfer at master

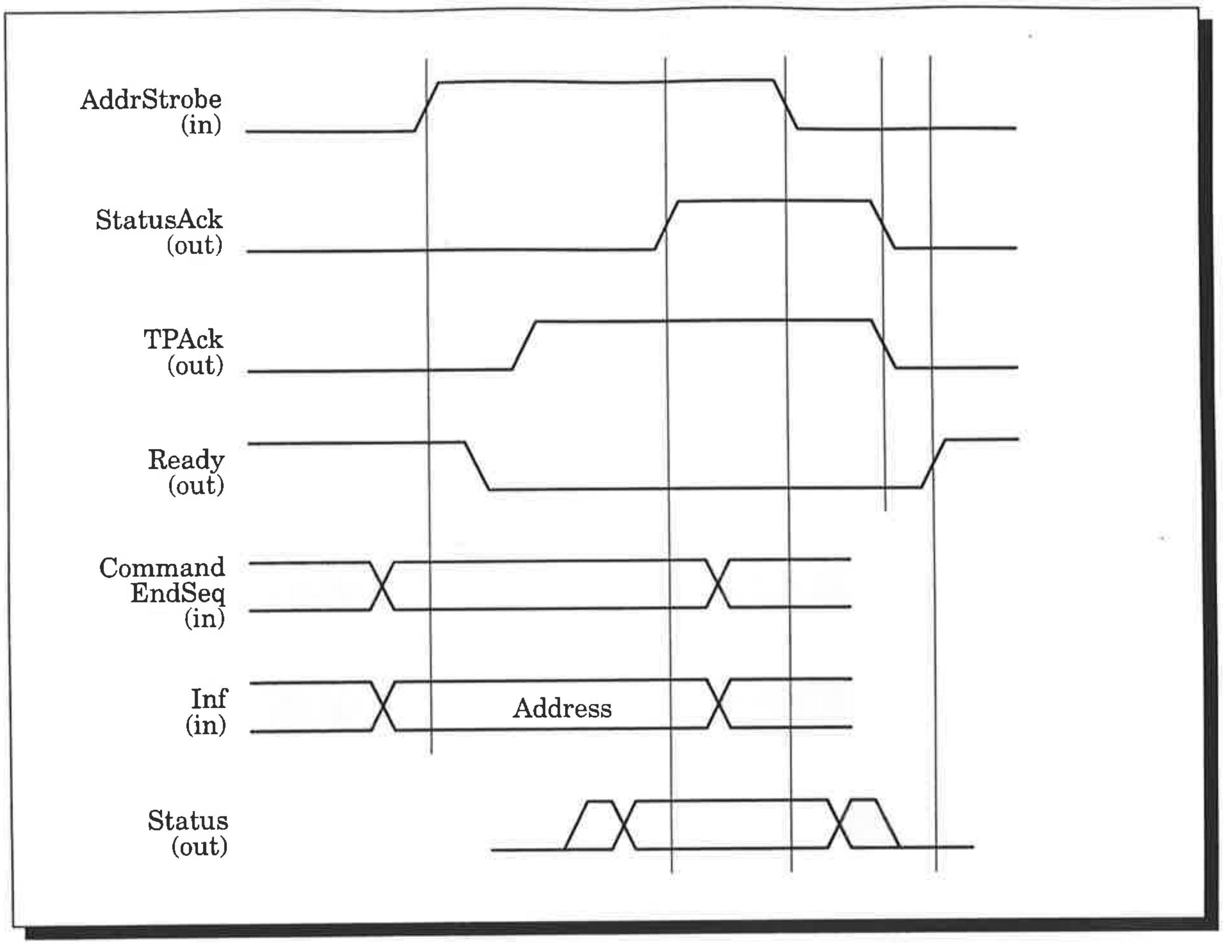


Figure A-3. Address transfer at slave

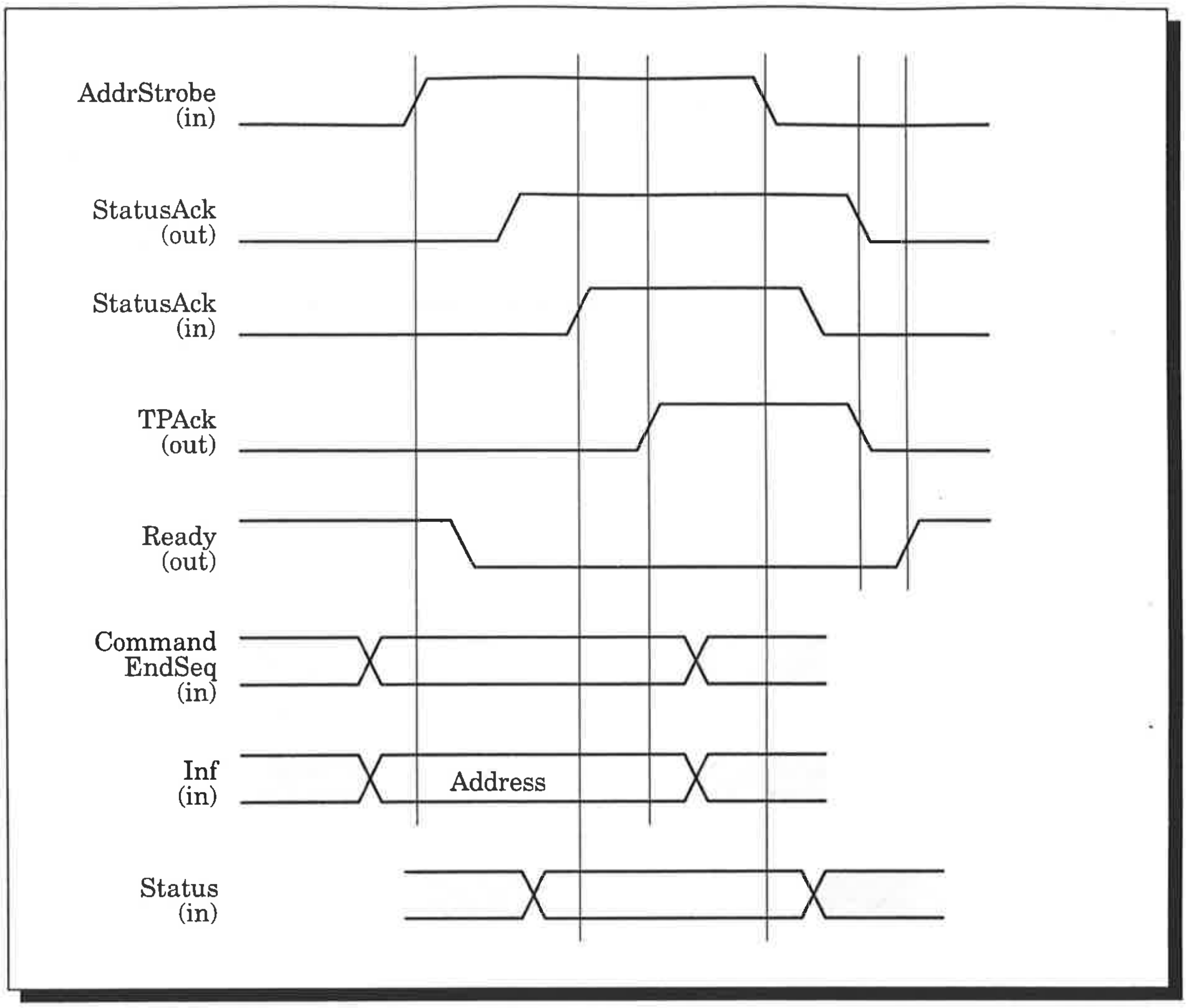


Figure A-4. Address transfer at third party

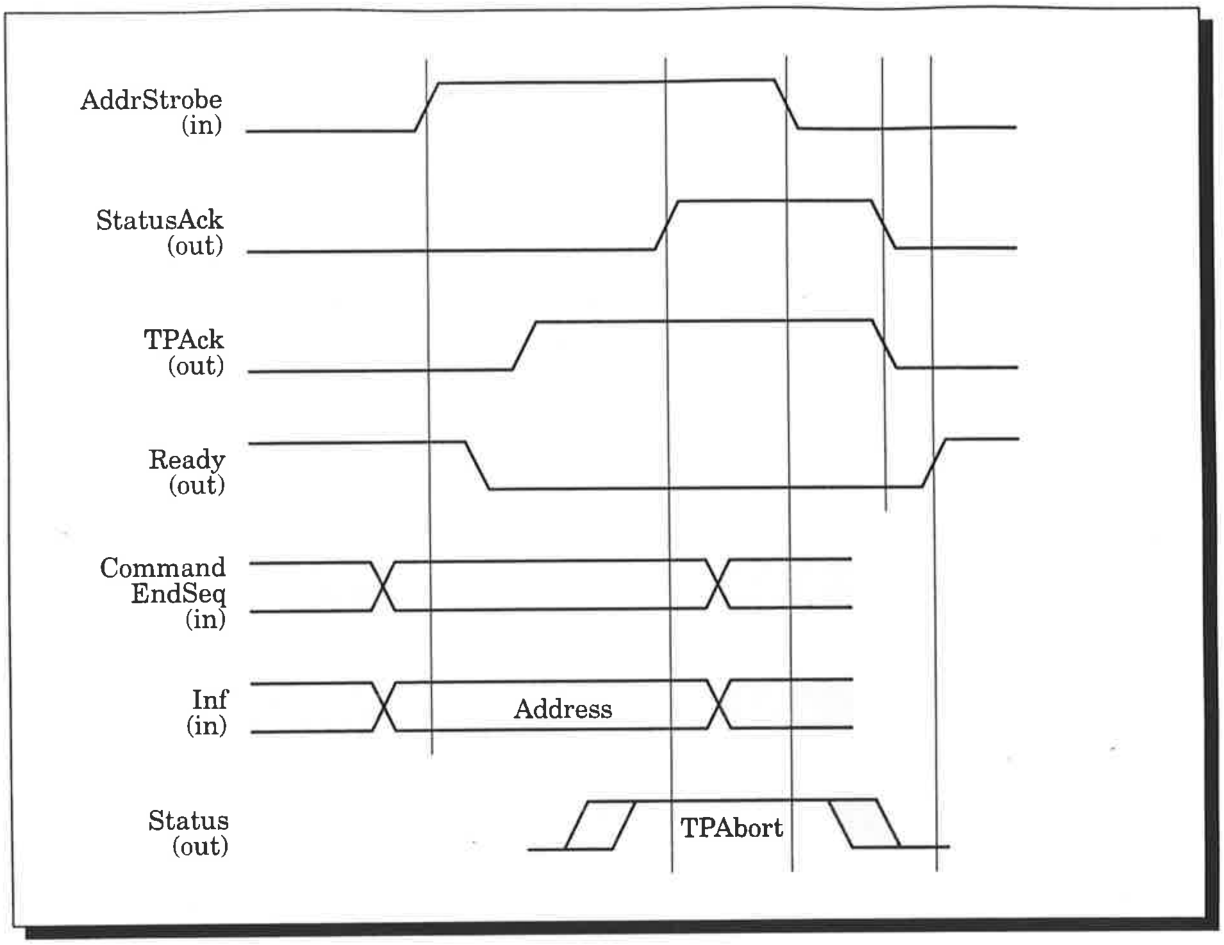


Figure A-5. Address transfer abort at third party

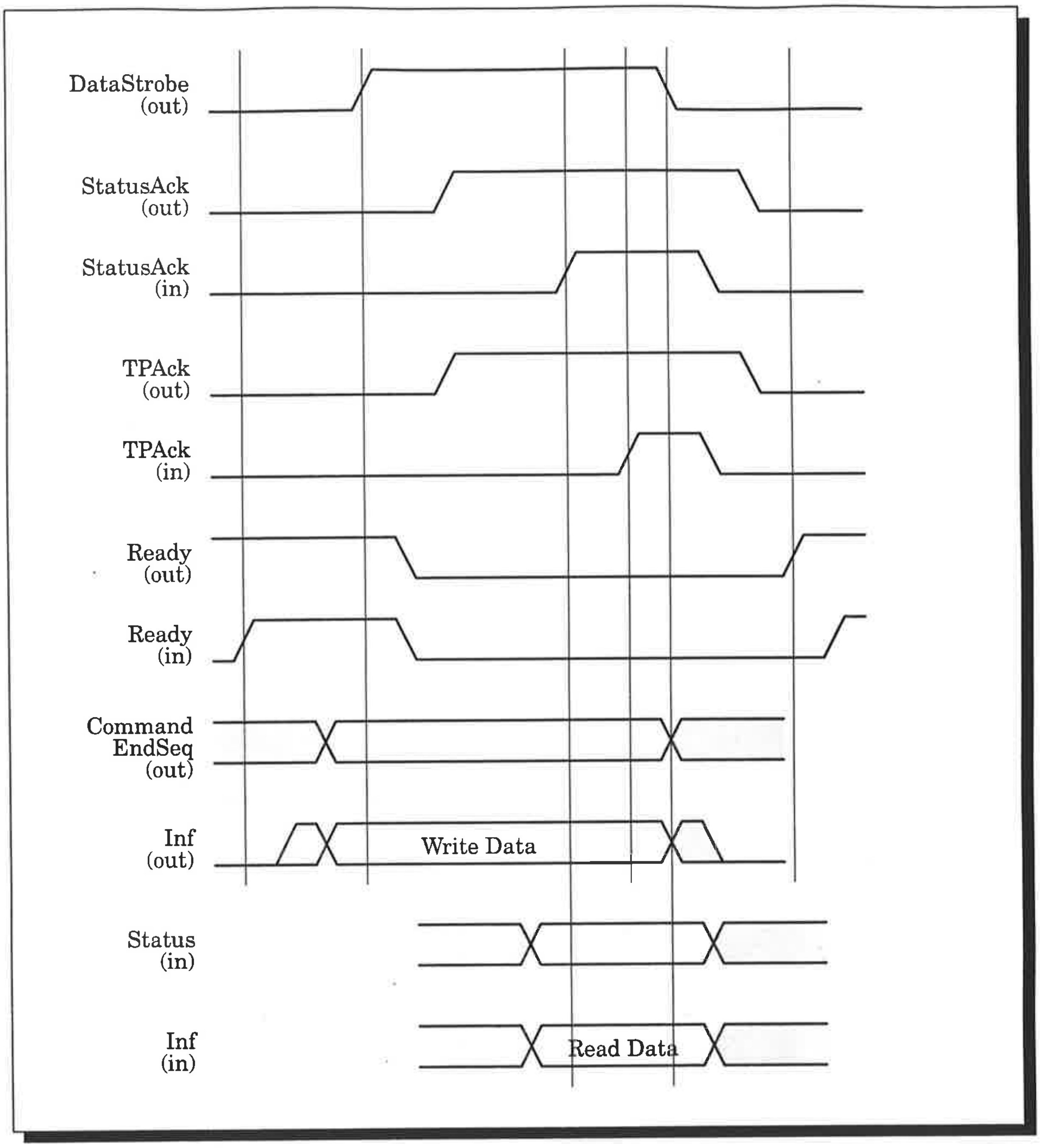


Figure A-6. Non-cachable data transfer at master

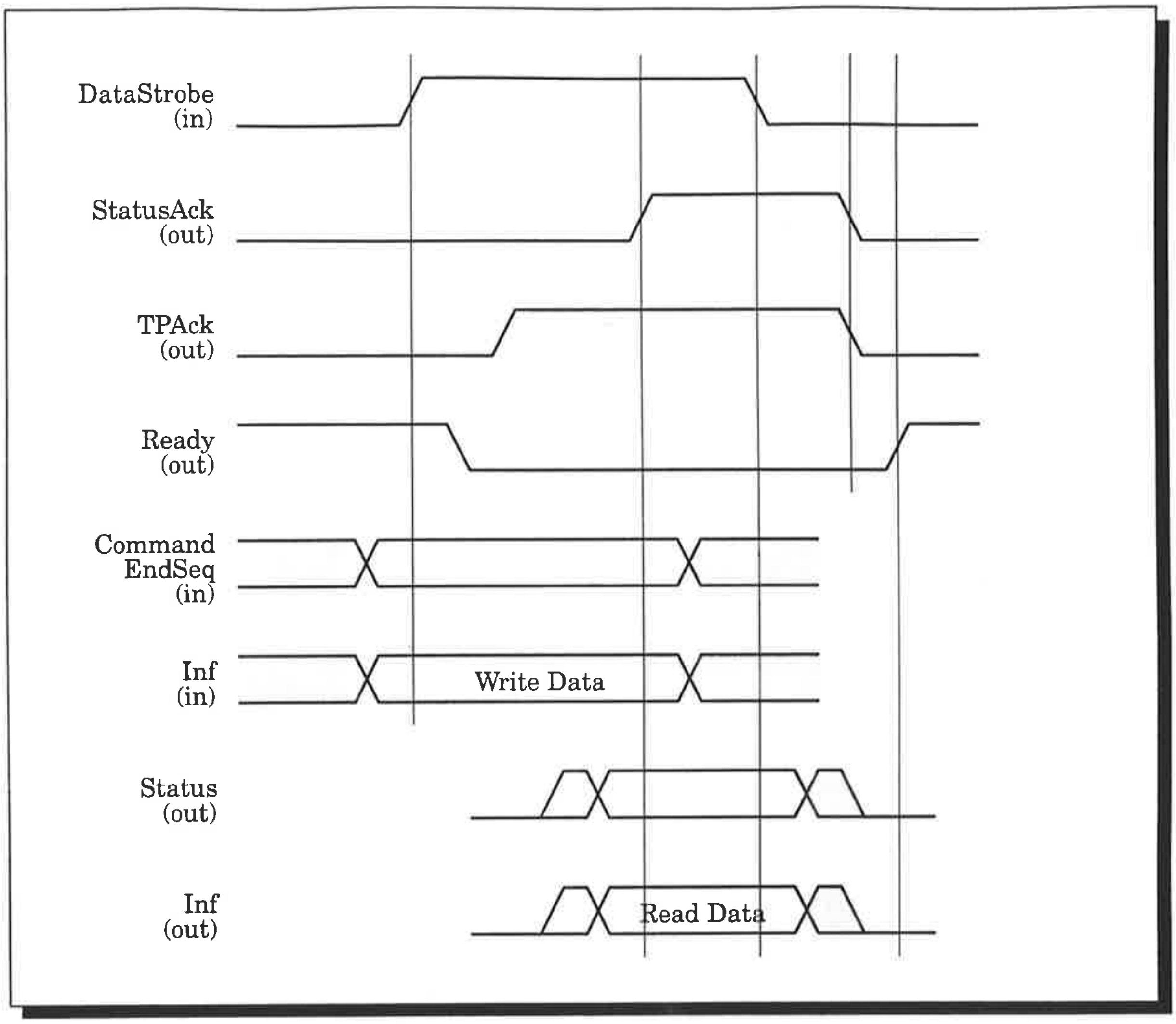


Figure A-7. Non-cachable data transfer at slave

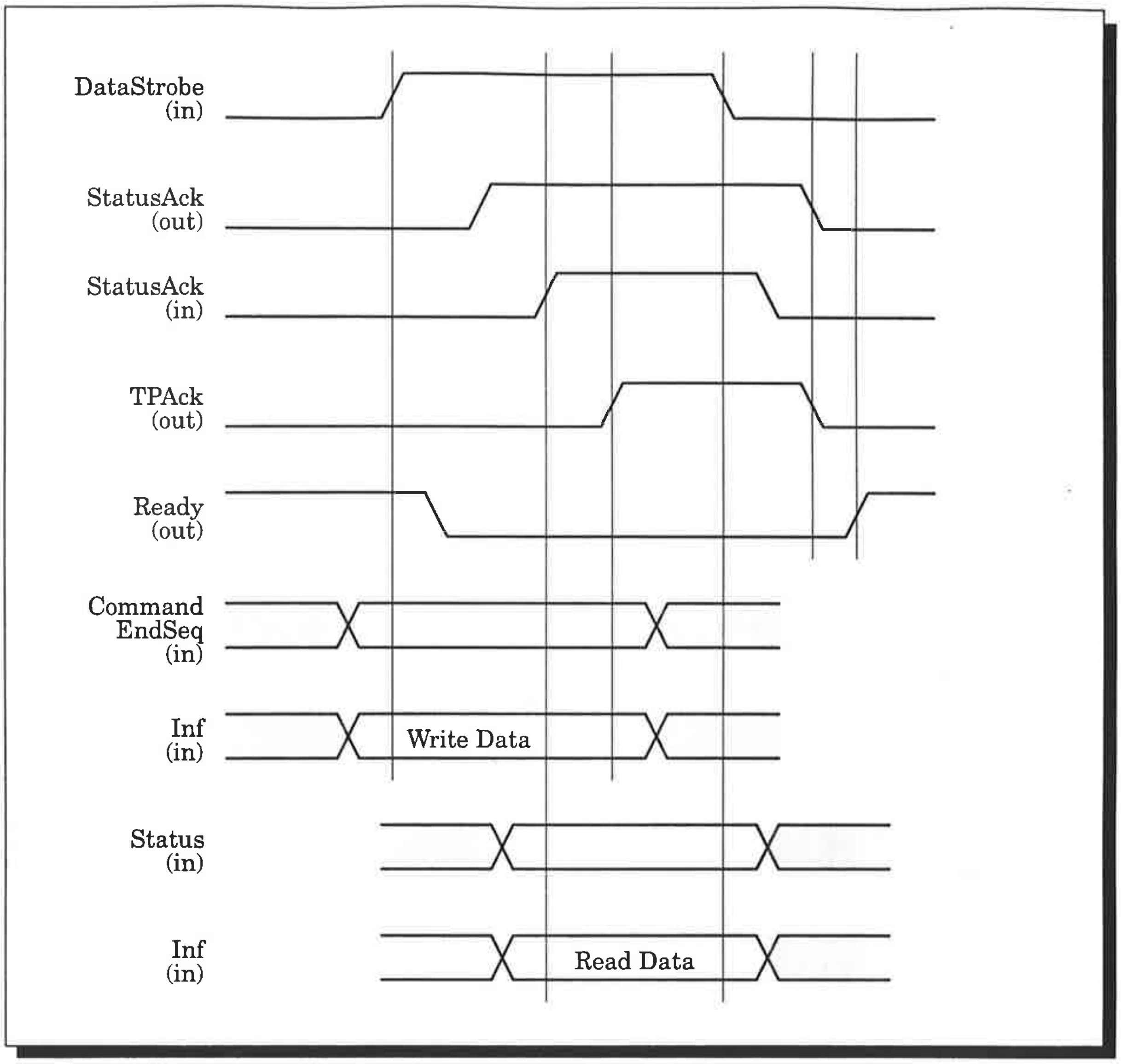


Figure A-8. Non-cachable data transfer at third party

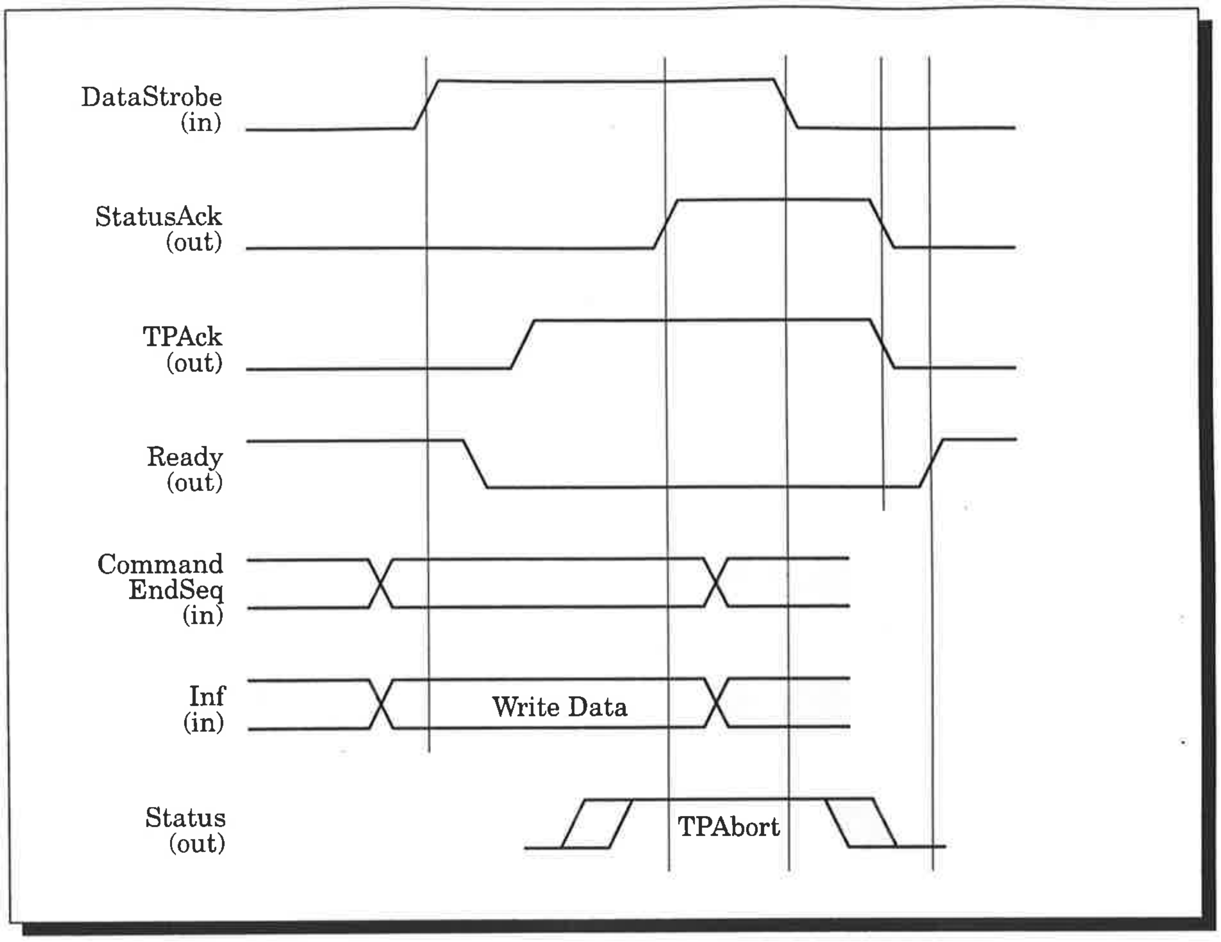


Figure A-9. Non-cachable data transfer abort at third party

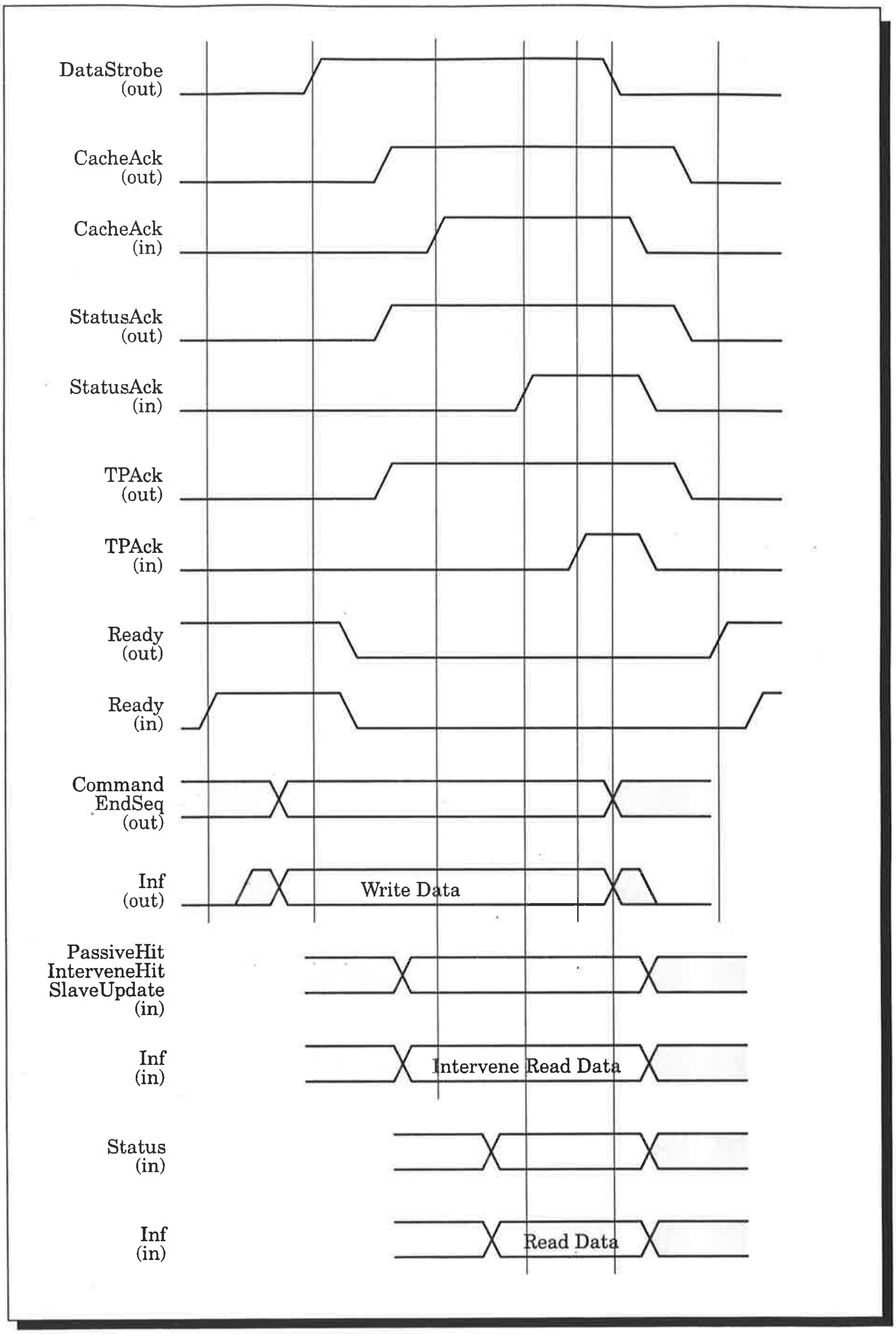


Figure A-10. Cachable data transfer at master

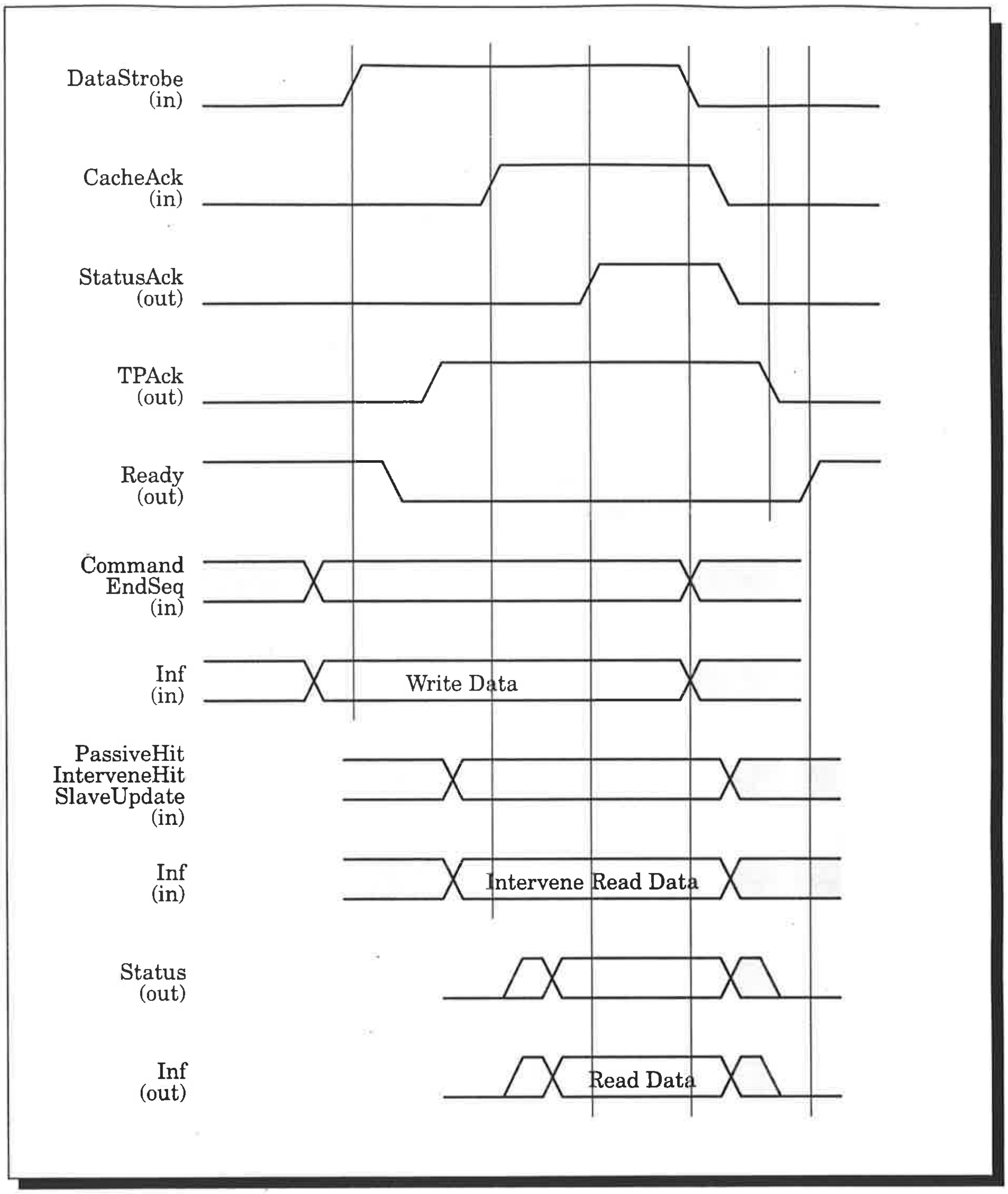


Figure A-11. Cachable data transfer at slave

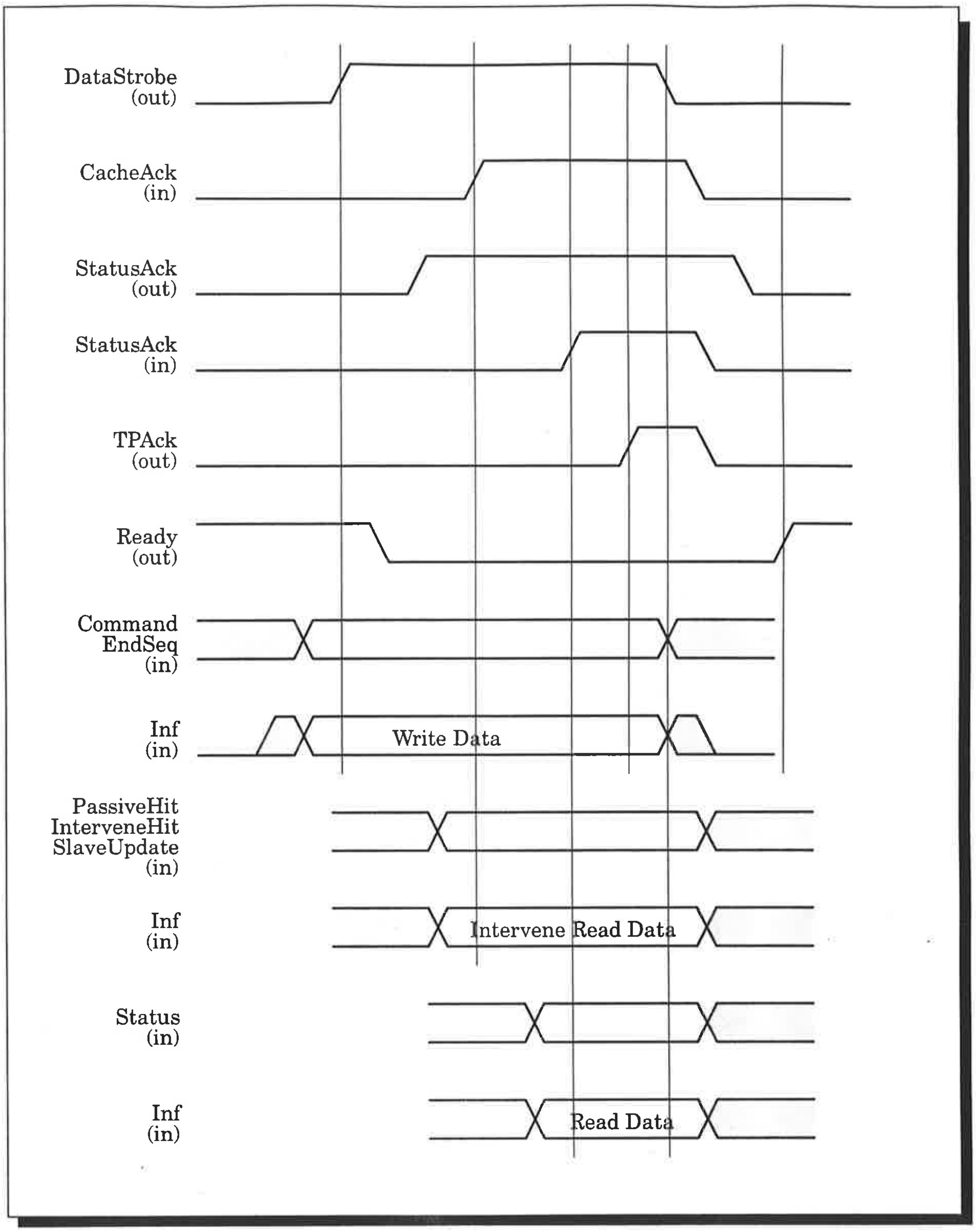


Figure A-12. Cachable data transfer at third party

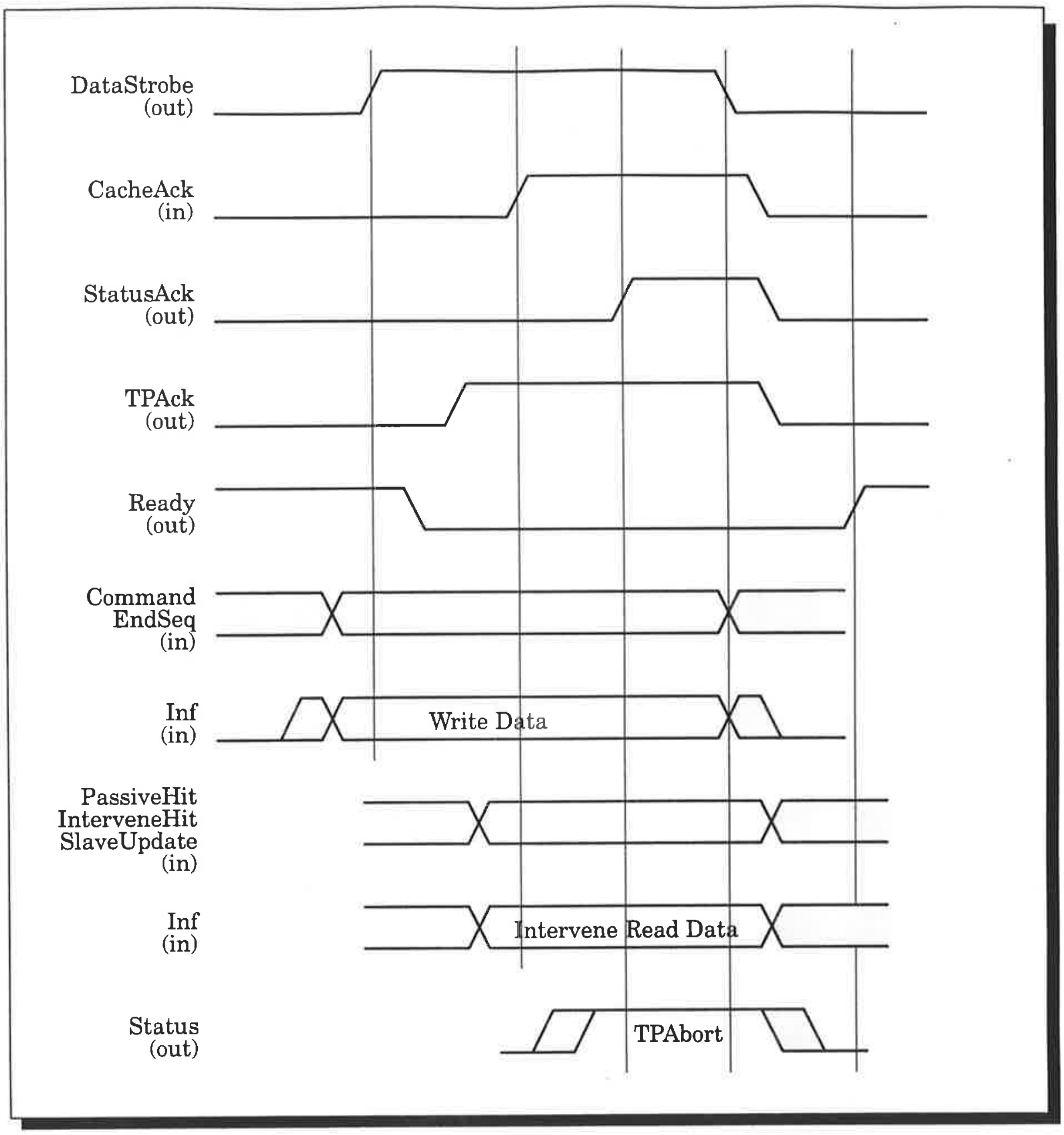


Figure A-13. Cachable data transfer abort at third party

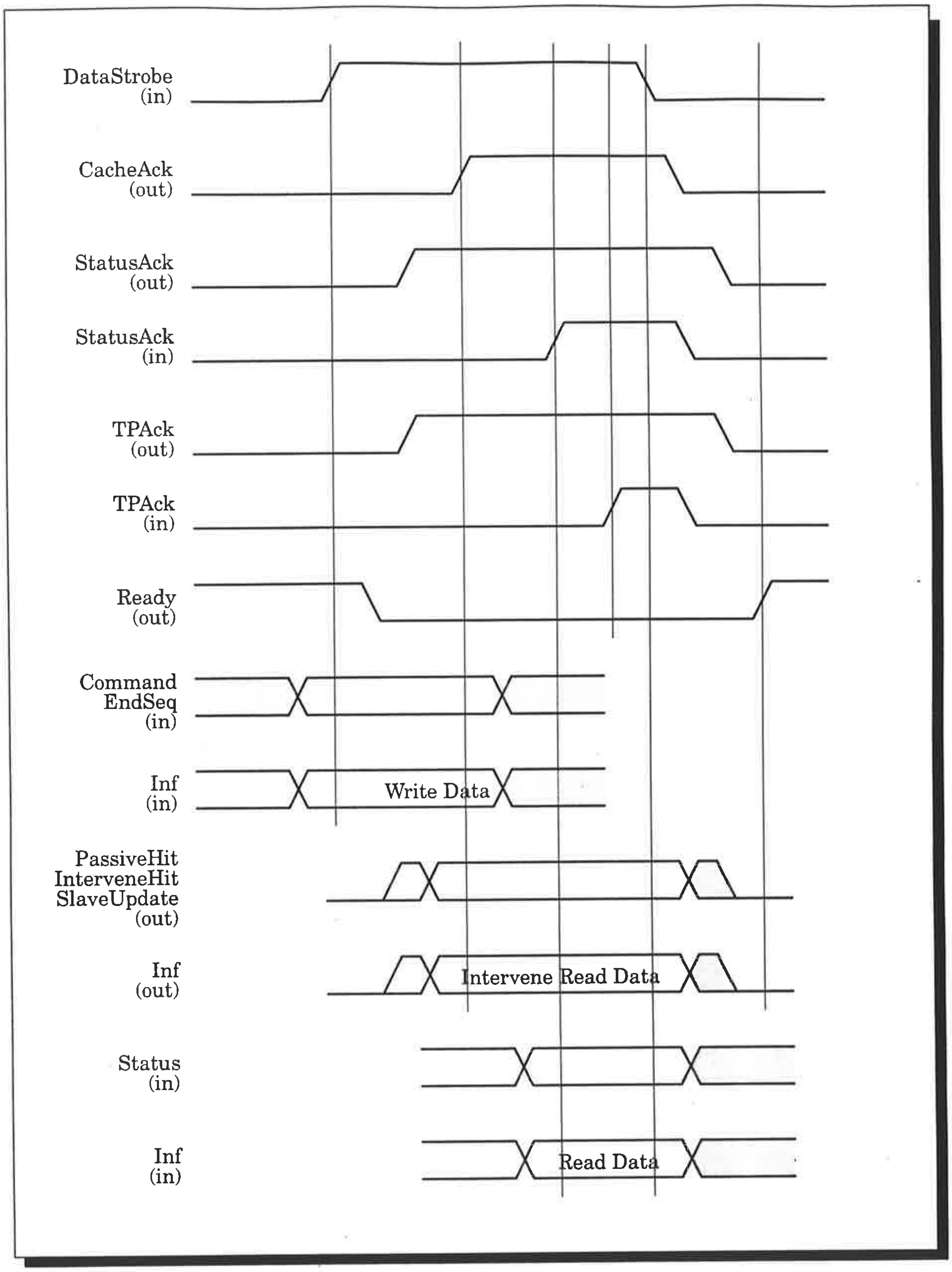


Figure A-14. Cachable data transfer at cache

Appendix B
The Leopard-2 Bus Arbitration Protocol

This appendix describes the arbitration protocol designed for use in the Leopard-2. It is based on that specified in the IEEE Futurebus standard [31], but is considerably simplified, allowing for significantly faster operation.

B.1 Arbiter and Protocol Description

A module which is to be involved in the arbitration process must include an arbiter with the interfaces illustrated in Figure B-1. The pins AP, AQ and AR are the synchronization signals used to sequence the arbitration protocol. The GA signal encodes the geographical address (slot number, between 1 and 24) for the module in the backplane. The signals REQ(24:1) are used by the arbiter to signal a request for access to the bus. A module in slot *i* uses REQ(*i*) to signal its request. The reset signal, RE, is

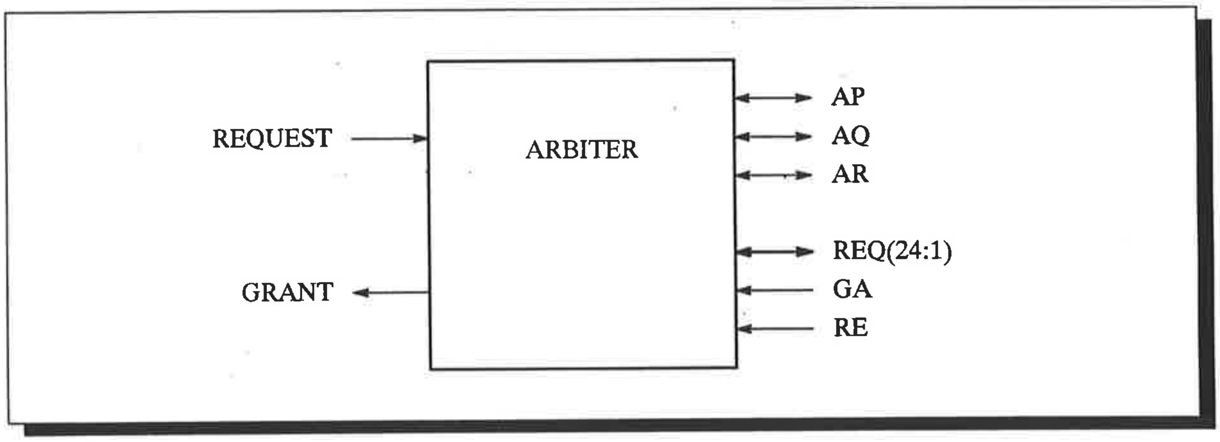


Figure B-1. Arbitration circuit connections.

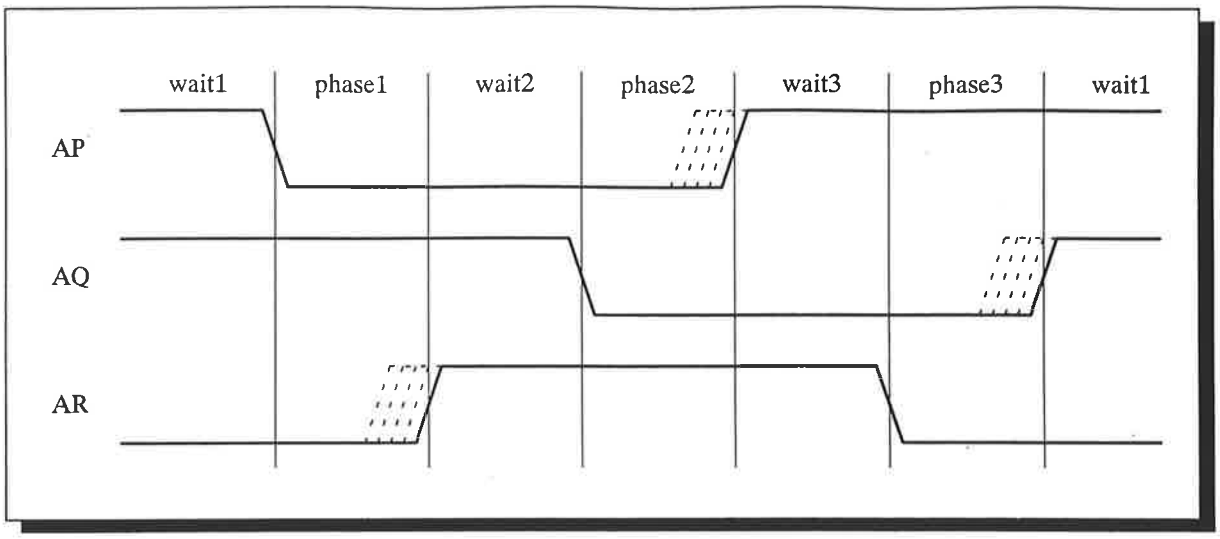


Figure B-2. Arbitration synchronization.

used to initialize the arbiter. The module uses the REQUEST signal to request access to the bus. When access is granted, the arbiter asserts the GRANT signal. The module then proceeds to use the bus, and when it has finished, it negates REQUEST. The arbiter responds by negating GRANT, after which a new request may be signalled.

The three synchronization signals, AP, AQ and AR, are active-low open-collector signals driven by all of the arbiters in a system. If any module asserts one of the signals (by pulling it low), that signal is in the asserted state for all modules. It is only in the negated state when it is released by all modules (and so pulled high by the bus terminator). This property is used to implement the three phase synchronization scheme, as illustrated in Figure B-2.

Initially, AP and AQ are released by all modules, and AR is asserted by all modules, and this state is called *wait1*. Any module may initiate state *phase1* by asserting AP, after which all modules may assert AP. When any module has completed processing for phase1, and after it has asserted AP, it may release AR. Only when all modules have released AR does the bus line become negated, and state *wait2* is entered. This process is repeated, cycling through the synchronization signals as shown in Figure B-2, until state wait1 is reached again.

The arbitration protocol is specified formally in VHDL in Section B.2 below, and operates as follows. Initially, after a reset operation, the synchronization signals are in state wait1, and there is no current master. When an arbiter which is not current master receives a bus request from its client module, it initiates state phase1 by asserting AP. All arbiters, when they detect the start of state phase1, also assert AP, and latch the state of their client request signals. Arbiter *i* drives signal REQ(*i*) with the latched value. When an arbiter has completed this operation, it releases AR. When all arbiters have completed the operation, state wait2 is entered.

When each arbiter detects state wait2, it asserts AQ to enter state phase2, and uses the values on REQ(24:1) and the knowledge of the current master to determine the master-elect. A round-robin allocation policy is used, where the master-elect is the requesting master with the least geographical address higher than the current master (modulo 25). When each arbiter has determined the master-elect, it releases AP. When all arbiters have completed the operation, state wait3 is entered.

If an arbiter detects state wait3 and there is no current master (after reset), it immediately initiates state phase3 by asserting AR. Otherwise, all arbiters wait for the current master to finish its use of the bus, after which it initiates state phase3. When state phase3 is detected, all arbiters also assert AR and make the master elect the current master. They then release AQ to signal completion of bus hand-over. The new current master, as soon as it has acquired that status, may start to use the data transfer bus. When all arbiters have completed the hand-over operation, state wait1 is entered. All arbiters remain in this state until a new request from a module other than the current master is detected. While in state wait1, the current master may regain tenure of the bus immediately, without having to arbitrate, so long as there are no other requests from other masters.

B.2 VHDL Specification of the Arbitration Protocol

The package arbitration_bus defines the types used in the arbiter specification. The type bus_signal represents an active low signal on the bus with only one driver. The subtype resolved_bus_signal is a resolved signal type used to represent active low signals which may have more than one driver. The type REQ_array is used for the request signals on the arbitration bus. GA_range specifies the allowed values for geographical (slot) addresses. The value 0 is reserved, and does not refer to any slot number. Finally, bus_settling_delay is the maximum round trip propagation delay specified in the Futurebus standard.

```
package arbitration_bus is
    type bus_signal is ('L', 'H');
    type bus_signal_vector is array (natural range <>) of bus_signal;
    function resolve_bus(drivers : bus_signal_vector) return bus_signal;
    subtype resolved_bus_signal is resolve_bus bus_signal;
    type REQ_array is array (24 downto 1) of resolved_bus_signal;
    subtype GA_range is integer range 0 to 24;   -- 0 reserved for no master
    constant bus_settling_delay : time;   -- time for signal driven on bus to settle
end arbitration_bus;
```

The body of the arbitration_bus package contains the specification of the bus signal resolution function. If performs a wired-or function, producing a low value (asserted) if any of the drivers are low, otherwise it produces a high value (negated). The value for the settling time constant is also specified in the body.

```
package body arbitration_bus is
    function resolve_bus(drivers : bus_signal_vector) return bus_signal is
        variable result : bus_signal := 'H';
    begin
        for i in drivers'range loop
            if drivers(i) = 'L' then
                result := 'L';
            end if;
        end loop;
        return result;
    end resolve_bus;
    constant bus_settling_delay : time := 25 ns;
end arbitration_bus;
```

The operation of the arbitration protocol is specified by the operation of each of the arbiters that participate in the protocol. The following declaration of the entity arbiter corresponds to the arbiter module shown in Figure B-1. The request and grant signals attach to the client processor. The processor requests bus tenure by asserting request. When it detects grant asserted, it may proceed to use the bus. When it is complete, it negates request, and waits until grant is negated. It may then make further requests.

```
use work.arbitration_bus.all;
entity arbiter is
    port (  request : in boolean;
            grant : out boolean;
            AP, AQ, AR : inout resolved_bus_signal;
            REQ : inout REQ_array;
            GA : in GA_range;
            RE : in bus_signal  );
end arbiter;
```

The architecture body of the arbiter specifies the behaviour in terms of a number of interacting processes. The internal signal current_master encodes the geographical address of the module that is currently granted bus tenure. The signal master_elect encodes the geographical address of the master that wins the arbitration process. The function next_master determines the geographical address of the module to be allocated tenure next, given the current master and the vector of requests. This function implements the round-robin policy. The process initialization sets the synchronization state to wait1, and the current master to 0, indicating that no master has tenure. The process acquire specifies that an arbiter whose client process requests the bus while not already current master starts the synchronization cycle by initiating the change to phase1. The process make_request specifies that when an arbiter detects the beginning of state phase1, it uses the current state of the client request to assert or negate its bus request signal. The non-deterministic decision point inherent in all arbiters is implied by this operation. The arbiter may take an indeterminate amount of time to decide whether the request input is asserted or negated when phase1 begins. The process elect specifies that when all arbiters have decided upon their requests, a new master will be elected based on the round-robin policy. The process tenure_release then specifies that further progress of the synchronization cycle is delayed until the current master has completed its bus tenure (at which time its grant signal is negated). However, if there is no current master (ie., during the first cycle), the synchronization cycle continues. The process hand_over specifies that when the current master has released the bus, the elected master becomes the current master, and its grant is asserted. The synchronization cycle is then complete, and returns to state wait1. The process release specifies that if the client process negates the request signal, and there are no other requests being arbitrated (indicated by still being in state wait1), the grant signal is negated, but the module remains current master. The process reacquire specifies that a current master may resume tenure without having to re-arbitrate, provided no other requests are being arbitrated.

```
architecture specification of arbiter is

    signal current_master : GA_range;        -- GA of current bus master
    signal master_elect : GA_range;          -- GA of elected bus master

    function next_master(current_master : in GA_range;
                         REQ : in REQ_array) return GA_range is
        variable candidate : GA_range;
    begin
        candidate := current_master + 1;
        if candidate = 25 then
            candidate := 1;
        end if;
        loop
            if REQ(candidate) = 'L' then
                return candidate;
            else
                candidate := candidate + 1;
                if candidate = 25 then
                candidate := 1;
            end if;
        end if;
    end loop;
end next_master;
```

```
begin   -- specification
    initialization : process (RE)
    begin
        if RE = 'L' then
            AP <= 'H';   AQ <= 'H';   AR <= 'L';   -- initially in state wait1
            current_master <= 0;                   -- bus allocated to no master
        end if;
    end process initialization;

    acquire : process
    begin
        wait until AP = 'H' and AQ = 'H' and AR = 'L'   -- wait1
                        and current_master /= GA and REQUEST;
        AP <= 'L';   -- start phase1
    end process acquire;

    make_request : process
    begin
        wait until AP = 'L' and AQ = 'H' and AR = 'L';   -- phase1
        AP <= 'L';
        if REQUEST then
            REQ(GA) <= 'L';
        else
            REQ(GA) <= 'H';
        end if;
        AR <= 'H' after bus_settling_delay;   -- wait2
    end process make_request;

    elect : process
    begin
        wait until AP = 'L' and AQ = 'H' and AR = 'H';   -- wait2
        AQ <= 'L';   -- start phase2
        master_elect <= next_master(current_master, REQ);
        AP <= 'H' after bus_settling_delay;   -- wait3
    end process elect;

    tenure_release : process
    begin
        wait until AP = 'H' and AQ = 'L' and AR = 'H'   -- wait3
                        and ( ( current_master = GA and not REQUEST )
                            or current_master = 0 );
        GRANT <= false;
        AR <= 'L';   -- start phase3
    end process tenure_release;
    hand_over : process
    begin
        wait until AP = 'H' and AQ = 'L' and AR = 'L';    -- phase3
        AR <= 'L';
        current_master <= master_elect;
        GRANT <= current_master = GA;
        AQ <= 'H' after bus_settling_delay;    -- wait1
    end process hand_over;

    release : process
    begin
        wait until AP = 'H' and AQ = 'H' and AR = 'L'    -- wait1
                        and current_master = GA and not REQUEST;
        GRANT <= false;
    end process release;

    reacquire : process
    begin
        wait until AP = 'H' and AQ = 'H' and AR = 'L'    -- wait1
                        and current_master = GA and REQUEST;
        GRANT <= true;
    end process reacquire;

end specification;
```

A Behavioural Specification of Cache Coherence

P.J. Ashenden† and C.D. Marlin†

Multiprocessor systems with shared memory are of increasing interest, because of their flexibility, incremental expandability, and potentially low cost. To reduce bus contention and to improve memory access time, such multiprocessor systems commonly incorporate a memory cache per processor. The use of memory caches leads to the need to ensure that the contents of the caches are coherent with each other and with the shared memory; this is the so-called *cache coherence* problem. A number of strategies have been proposed to overcome this problem, but little is known about their advantages and disadvantages.

This paper contributes to the study of cache coherence strategies by proposing a formal model of cache coherence. This model, which is an information structure model, is described and its application illustrated by outlining how it can be used to describe the cache coherence strategy used in the Futurebus standard.

Keywords and Phrases: **cache coherence strategies, information structure models, multiprocessor computer systems, Futurebus.**
CR Categories: **B.3.2, B.3.3, C.1.2.**

1. INTRODUCTION

A multiprocessor computer architecture which has received much attention recently is the bus-based *symmetric multiprocessor.* It consists of a pool of homogeneous processors connected via a system bus to a globally shared memory. This basic configuration may also be augmented with processors and controllers for particular functions such as I/O interfacing, graphics processing and graphics display. Several examples may be found, both as commercially manufactured systems [e.g. Sequent Balance (Fielland and Rodgers, 1984) and Symmetry (Manuel, 1987), Encore Multimax (Anzelmo et al., 1985) and the DEC VAX 8000 Series multiprocessors (Digital Equipment Corporation, 1985), and as experimental systems for research, e.g. SPUR at U.C. Berkeley (Hill et al., 1986), and Leopard at University of Adelaide (Ashenden et al., 1987)]. The symmetric multiprocessor architecture has a number of important advantages for many applications. See Ashenden, Barter and Marlin (1987) for a discussion of these.

From consideration of processor-to-memory access bandwidths and bus data transfer bandwidths, it is clear that a data cache on each processor is crucial to the successful operation of a symmetric multiprocessor. Without a cache, the bus would be a source of congestion, with each processor having to wait for access to code and data stored in shared memory. A first order approximation would indicate that the use of caches increases the number of processors which can effectively use a bus by a factor related to the cache hit-rate.

Of interest is the work undertaken by the IEEE Futurebus Standard Working Group (IEEE Standard 896.1, 1987). Until recently, there was no standard bus suitable for use in a symmetric multiprocessor implementation. Manufacturers and researchers resorted to their own custom bus designs [e.g. the Balance system bus, the Multimax Nanobus, and L-Bus (Ashenden and Knight, 1985) in the Leopard-1 system]. The problems with existing standard buses were that they assumed a monoprocessor as bus master, they had insufficient bandwidth, and they provided no support for multiple caches. More recently introduced buses [such as Multibus-II (IEEE Standard 1296, 1987) and VME (IEEE Draft Standard 1014, 1987)] have addressed the performance issues and allow symmetric multiple masters, but still provide no cache support. The Futurebus design, on the other hand, has addressed all of these problems, as will be described below.

The Futurebus Standard specification has been divided into two sections. IEEE Standard 896.1 (1987) defines the mechanical and electrical details, and specifies the basic protocol mechanisms for bus transactions. The second section (called P896.2, currently being drafted) will define the next level of protocol, including message passing formats, event notification, and the cache coherence protocol. The first author has been involved in development of both sections, concentrating on the specification of cache coherence in P896.2. The issue of how to specify coherent cache behaviour has been given much attention by the Working Group, and the approach promoted by the authors is presented in this paper.

2. CACHE COHERENCE

The model of a symmetric multiprocessor used to consider the behaviour of caches is illustrated in Figure 1. It consists of a shared memory accessible via the system bus and a number of caches, each of which serves a client. (Typically the clients would be processors.) Data in shared memory is

† *Department of Computer Science, University of Adelaide, GPO Box 498, Adelaide, SA, 5001. This paper was presented at the Eleventh Australian Computer Science Conference at the University of Queensland in Brisbane, Queensland in February 1988.*

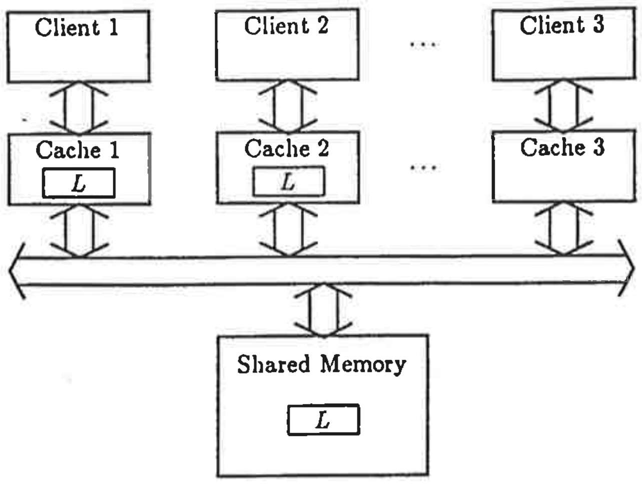


Figure 1. Data sharing between multiple caches.

treated as an array of equally sized contiguous blocks called *lines.* The service provided by each cache is fast access to lines of shared memory data. This is achieved by storing locally a copy of those lines expected to be used by the client. [See Smith (1982) for a description of cache design parameters.] The requirement placed on the cache service is that, at any time, all clients are given the same image of the data in shared memory. This property is called *cache coherence.*

To illustrate the way in which caches can interact to maintain coherence, suppose the caches in Figure 1 are copy-back caches, and that tasks executing on the three clients are sharing access to data contained in line *L*. The data is stored in shared memory, and Caches 1 and 2 each have a copy, by virtue of having read the line from shared memory.

Suppose Client 1 wishes to modify *L*. Given that there is another cached copy of *L*, Cache 1 cannot simply modify its local copy, since that would produce a violation of coherence. Amongst the options available are that it *broadcast* the new data to all other caches (and possibly to shared memory as well), or that it broadcast a signal for all other caches to *invalidate* their copies of *L*. For the purpose of this illustration, assume it does the latter. In this case, Cache 1 will then contain the only current version of *L* (call if *L'*) with shared memory still containing the old version.

Now suppose Client 3 wishes to read the data. If Cache 3 were to fetch the data from shared memory, it would fetch the old version. It is necessary for Cache 1 to become involved in the transaction to supply the current version *L'*. It can do this by disabling the shared memory, and *intervening* to *supply L'* in its place. Alternatively, it could supply *L'* in place of shared memory, and cause shared memory to make use of the data on the bus to update the primary copy (i.e., shared memory does a write). This is called *reflection.*

The way in which caches respond to client requests and bus transactions in order to maintain coherence is called a *cache coherence strategy*. There have been several coherence strategies published in the literature and implemented in experimental systems over the past four years. These include Goodman's *write-once* strategy (Goodman, 1983), Papamarcos' *Illinois* strategy (Papamarcos and Patel, 1984), the *Berkeley ownership* strategy used in SPUR (Katz et al., 1985), and the XEROX Dragon and DEC Firefly strategies (Archibald and Baer, 1986).

These strategies are typically defined in an informal manner. Comparisons between them have been made in an informal descriptive manner, using analytic models (Vernon and Holliday, 1985) and using the results of simulation studies (Archibald and Baer, 1986). The analytic models used differ from the model to be presented in this paper in that they are aimed at estimating performance of systems incorporating the modelled strategies, whereas our model is aimed at precise specification of strategies in general.

All of the above strategies have some points in common. Firstly, they all augment the usual *valid/invalid* state bit of a cache entry with additional status bits to reflect further attributes of a line (e.g. degree of sharing, whether the line has been modified, etc.). Secondly, they all require caches to monitor bus transactions, and possibly to change line state if they have a copy of the data being accessed by the transaction. Thirdly, they all assume special bus support for maintaining coherence. This support takes the form of additional transaction types beyond the usual read and write, such as invalidation, intervention and broadcast transactions.

Prior to the development of Futurebus, standard buses provided no protocol mechanisms for such things as notification of invalidation or intervention. Those proprietary buses developed to support particular cache coherence strategies only included the necessary mechanisms for their particular strategies. The goal of the Futurebus Committee was to provide protocol mechanisms to support all coherence strategies. This required investigation of the published strategies to determine the set of transactions required. As a result of this investigation, a preliminary model of coherent cache behaviour was formulated (Sweazey and Smith, 1986), and the basic bus protocol of IEEE Standard 896.1 was designed to support implementation of this model.

3. SPECIFICATION TECHNIQUES

The current work of the Cache Coherence Task Group of the P896.2 Working Group is to draft a specification of the behaviour of caches in a Futurebus system. The specification must ensure that any conforming implementation maintains coherence, and must be flexible enough to include published coherence strategies as subset implementations. A problem to which the Task Group has given much consideration is the selection of a specification language. This is a significant problem, since the solution has bearing on the effectiveness of the specification as a standard. It must combine precision and completeness with intelligibility by its intended audience.

The specification of cache coherence strategies has a number of parallels with the specification of programming language semantics. The same range of possible specifica-

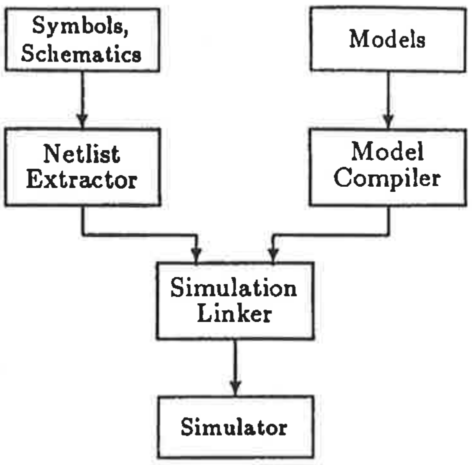


Figure 2. Helix system organisation.

tion techniques applies, from very informal to highly formal; as with programming languages, the most common techniques are at the informal end of this spectrum. The motivation for the development of formal models of cache coherence strategies is also similar to that motivating the development of formal models of programming language semantics, and comes in three parts:

— to help in the process of understanding (and comparing) cache coherence strategies,
— to provide a less ambiguous way to describe a particular cache coherence strategy, and
— to assist with proving various properties of a particular cache coherence strategy.

In view of these similarities, it is not surprising that the sorts of models which are useful in the formal description of programming language semantics are also useful in the formal description of cache coherence strategies. At the informal end of the specification spectrum, a natural language (e.g. English) may be used. However, its apparent advantage of being understood *a priori* by its intended audience is, in fact, its weakness. Natural languages are learned by example, and sentences may be interpreted differently by different people. The diversity of interpretation and nuance which makes poetry and other artistic literature possible precludes its use for formal documents. Where precise natural language is required, documents are circumlocutious, laboured, and often ambiguous and opaque to understanding, despite the best efforts of their authors. Furthermore, concepts expressed in natural language are not amenable to verification through analytical or mathematical means.

At the highly formal end of the spectrum of specification languages are formal mathematical notations, such as abstract temporal logic [e.g., LOTOS (ISO/TC97/SC21 DP-8807, 1985)] and Petri nets (Peterson, 1977). These achieve precision by virtue of a rigorous mathematical basis, and hence properties of the concept being specified can be proven mathematically. However, such notations are often inappropriate, since they rely on the proficiency of the authors in expressing the concept in the formal terms, and are not easily understood by the large number of system designers who must read the specification. In addition, it is not always clear how to test a physical system for conformance to a specification written in such a notation.

A compromise between the above extremes is represented by the notion of an *information structure model* (Wegner, 1971). This kind of model is especially useful in the description of programming language semantics, where the state of a program is described by an information structure (essentially a data structure) and the semantics of a particular language feature is described by giving its effect on the information structure. For example, the information structure used by Basili (1975), in his description of the semantics of some language features for graph manipulation, is a collection of sets modelled as graphs. Similarly, Marlin and Oudshoorn (1985) use an information structure consisting of a collection of tables in their description of the data control aspect of programming languages.

An information structure model can also be used to describe the behaviour of coherent caches in a multiprocessor system. In this case, the state of each cache is represented by an information structure, and the changes in cache state are represented as transformations on the information structure. The contents of the information structure include the representation of attributes of lines of data (e.g. validity, sharing, etc.). The transformations represent the behaviour exhibited by a cache in response to transactions on the system bus, and to requests made by the corresponding cache client.

Using an information structure model as a specification has several advantages, particularly in the context of a standards document. Firstly, the information structure and the transformation operations can be expressed in a familiar "programming language" form. This makes it easier to produce the standard, and leads to a standard which is more intelligible to its intended audience.

Secondly, because of the programming language form, the specification of a system can be simulated. This requires an interpreter for the language, provision of some concrete representation of the information structure, and an environment for executing the transformation operations in response to some externally provided stimulus events.

Thirdly, as a result of a simulation, test vectors can be created, and subsequently used to verify conformance of an implementation to the specification. This is analogous to the use of validation suites to test programming language implementations for conformance to a semantic specification. The simulation can also be used as a reference implementation, being the arbiter in the case of disagreements between implementations.

A common criticism of specification languages in general is that they themselves need to be specified formally, and this is also true of specification systems based on information structures. However, the two components of an information structure model can themselves be specified in precise terms:

— The information structure can be described precisely using algebraic techniques for specifying abstract data types (Goguen, 1975; Goguen et al., 1977; Guttag, 1980 and Guttag et al., 1978); for illustrations of how this can be done see Friedel et al. (in preparation), Marlin and Oudshoorn (1985) and Oudshoorn and Marlin (in preparation).
— The transformations can be written in the notation of a programming language (either pre-existing or designed for the purpose) whose semantics can be specified formally using techniques such as denotational semantics (Tennent, 1976).

In this way the information structure model will use only primitives which have precise descriptions, thus ensuring that the model has firm foundations.

4. A BEHAVIOURAL SPECIFICATION USING HELIX

As part of the work for the IEEE P896.2 Cache Coherence Task Group, we have developed a behavioural specification of the Futurebus Cache Coherence protocol using the Helix simulation system (Silvar-Lisco Corporation, 1986). This system is part of a Computer Aided Engineering suite, and is a discrete event simulator specialised for modelling electronic systems.

The behavioural specification could be written using some other programming language, such as Pascal, Lisp, or a concrete form of such mathematical notations as temporal logic (e.g. Tempura (Moszkowski)). The requirement is that there be a formal semantic basis underlying the language used. Helix was chosen because its modelling language is specially designed for expressing behavioural interactions in complex electronic systems, and the runtime system provides most of the infrastructure required for managing a simulation and collecting results.

4.1 Overview of the Helix System

Figure 2 illustrates how the Helix simulation system is used. Graphical symbols representing components, and a schematic representing a circuit of interconnected symbols, are created using a graphics editor. A netlist extractor is used to determine the electrical connectivity drawn in the schematic, and to perform some validity checks on the circuit. For each component type to be simulated, a behavioural model is written using the Helix Hardware Description Language (HHDL). These models are then checked and compiled into an intermediate code. Next, the simulation linker is invoked to check for consistency between component symbols and models, and to create a simulator for the schematic.

The HHDL language is based on Pascal, augmented by constructs for representing component pins and concurrency constructs for implementing component actions in response to pin stimuli. An HHDL program firstly defines *nettypes*, which are (almost) arbitrary Pascal data types used to represent values passed on signal nets between component pins. Then, for each component type in the circuit, a *comptype* is defined. This consists of the specification of the pins, naming the nettypes they may connect to, some local state expressed in the form of local variables, a collection of *subprocesses*, and a main body for initialisation. The subprocesses are bodies of code which are activated when specified conditions occur; typically, the conditions are changes of values on input pins.

```
module flipflops;
const Tpd = 10;
nettype lognet = (Unk, Z, Lo, Hi);
comptype Dff;
  inward D, CLK : lognet;
  outward Q : lognet;
  subprocess sample :
    upon (CLK=Hi) and (recall(CLK)=Lo)
    check CLK do
      begin
      case D of
        Lo, Hi : assign D to Q delay Tpd;
        Unk, Z : assign Unk to Q delay Tpd;
        end;
      end; (* sample *)
  begin (* Dff *)
  Q := Lo;
  end; (* Dff *)
```

Figure 3. HHDL model for a D-type flip-flop.

Figure 3 is an example of HHDL code for a D-type flip-flop (called *Dff*), contained in a separately compiled module called *flipflops*. The net-type *lognet* defines the four-state type commonly used for logic simulation, with values for *unknown*, *high impedance*, *low* and *high* logic levels, respectively. Comptype *Dff* has two input pins, for data and clock, and a data output pin. Its initialisation body resets the output to the low logic level. The subprocess *sample* is sensitive to changes in the value on the *CLK* input (indicated by the phrase "check CLK"), and is activated when the new value is *Hi* and the previous value was *Lo*. When activated, a new value is passed onto the output pin, based on the current value of the data input pin.

The simulator created by the simulation linker is a program which contains an instance of a component model for each use of the component in the schematic. A run-time environment is provided which represents the signal nets connecting component instances, schedules updates of nets on a simulation timeline, and activates comptype instances at the required timepoints. For example, the *assign* statements in Figure 3 cause the run-time system to schedule an update of the net connected to the *Q* output of the flip-flop at a time point *Tpd* units after the current activation. The run-time system also collects a history of net updates when the program is run, and this history is used by formatting tools to create tabular or logic-analyser type displays of the simulation. Additional information about the behaviour of individual components in the circuit may be obtained by embedding trace write statements in the HHDL models. The simulator is controlled by a debugger-style command interface, providing single-stepping, breakpoints, and other similar facilities.

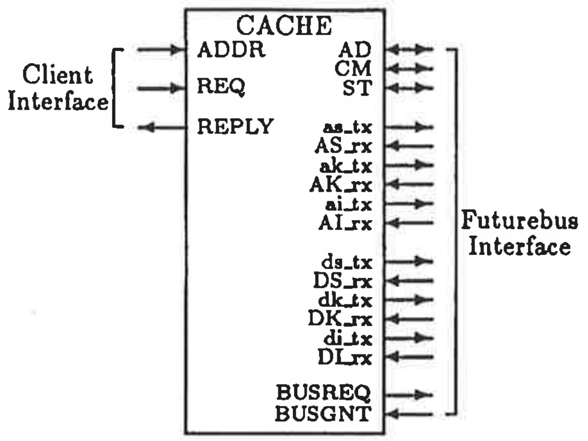


Figure 4. Symbol for a Futurebus Cache.

4.2 Specification of Coherence

The Helix system can be readily used to create an information structure model of cache coherence. This is done by defining a symbol for a cache, with pins to connect to a client and a Futurebus (see Figure 4), and specifying an HHDL comptype for the cache behaviour. The comptype contains local variables to implement the information structure representing the cache, and the transformation operations are encoded as comptype subprocesses.

One of the problems in designing any model is to determine which aspects of the system under consideration are actually relevant, and which can be ignored. In the case of modelling caches for the specification of coherence, the actual lines of data are not relevant; it is the line addresses and attributes that are important. For this reason, the information structure representing a cache does not contain any reference to the data which may be stored in a cache.

Another difficulty in model design is the temptation to incorporate details of some particular implementation. For example, a model might represent the cache storage as a small array of entries for storage of line tags and status bits. However, this approach would then require attention to the various cache design parameters (as mentioned in Smith, 1982) in order to specify cache behaviour. A better solution is to represent the attributes of each shared memory line, including the validity attribute.

Given the above two considerations, the information structure for a Futurebus coherent cache is implemented as shown in Figure 5. The store is represented by an array of status attributes, one entry for each line of shared memory data. The attribute *valid* indicates whether the line is stored in the cache's memory, *shared* indicates whether some other cache in the multiprocessor may also have a copy of the line, and *owner* indicates whether this cache is the owner of the line. Ownership means that the line has been updated without being written to shared memory, and implies an obligation to supply the data in response to bus read transactions, and to perform a copy-back or transfer of ownership at some stage. Note that lack of validity implies lack of the other two attributes.

```
type
  line_index_type = 0..nr_lines−1;
  status_type = (valid, shared, owner);
  line_type = record
    status : set of status_type;
    end;
  store_type = array [line_index_type]
                 of line_type;
  cache_config_type = record
    accept_broadcast,
    reflecting_owner, ... : boolean
    end;
var
  store : store_type;
  config : cache_config_type;
  connection_status : (unselected, selected,
                         intervening, reflecting);
  bus_hit, keep_copy, ... : boolean;
  (* other temporary variables ... *)
  ...
```

Figure 5. Information structure for a Futurebus coherent cache.

The variable *config* is used to represent the diversity of possible coherence strategies which must be encompassed by the Futurebus coherence protocol. It contains a set of switches which are used at decision points in the transformation operations to govern how the information structure is modified. Following it are state variables used to communicate between the different HHDL subprocesses which define the transformations. The remaining variables are temporary storage used to implement the transformations.

The HHDL subprocesses which implement the transformations on the information structure fall into two groups: the *snoop* group, in which subprocesses are activated in response to Futurebus transactions, and the *server* group, in which subprocesses are activated in response to client requests. To illustrate the way in which the transformations are implemented, the snoop group will be discussed in some detail. The interested reader is referred to the draft P896.2 document (IEEE Draft Standard 896.2) for the protocol relating to service of client requests.

The snoop group contains three subprocesses, invoked on the address beat, each data beat and the end beat, respectively. The address beat subprocess, outlined in Figure 6, implements the cache lookup function, checking whether the cache has a hit at the address of the bus transaction. It is sensitive to changes in the address strobe input (*AS_rx*), and is activated when this pin changes from *Lo* to *Hi*, and the cache is not acting as a bus master. The address and command information placed on the bus by the bus master is accepted for use in this beat and the subsequent beats comprising the transaction.

Next, the cache arbitrates for exclusive use of the line of data being accessed by the bus transaction. The arbitration is required, as the cache client may asynchronously request access to the same line, possibly modifying its attributes.

```
subprocess do_address_beat :
  upon not master and (AS_rx = Lo) and (recall(AS_rx) = Hi)
  check AS_rx do
    begin (* do_address_beat *)
    bus_line_index := RegToInt(AD[31..6]);
    bus_quadlet_index := RegToInt(AD[5..2]);
    bus_addr_command := CM;
    bus_acquire_mutex(bus_line_index);
    with store[bus_line_index], bus_addr_command do
      begin
      bus_hit := valid in status;
      if bus_hit then
        begin
        if IM and BC then
          if config.accept_broadcast then
            connection_status := selected;
        if (not BC) and (owner in status) then
          begin (* must reflect or intervene *)
          read_occurred := false;
          if config.reflecting_owner then
            begin
            connection_status := reflecting;
            all_dirty_reflected := false;
            assign Lo to di_tx;  assign Z to dk_tx;
            end
          else (* intervening owner *)
            begin
            connection_status := intervening;
            assign Lo to di_tx;  assign Lo to dk_tx;
            end;
          end;
        keep_copy := (connection_status in [selected, intervening, reflecting])
                     or ( (connection_status = unselected)
                          and not IM and config.keep_after_unselected);
        with bus_status do
          begin
          sl := connection_status = selected;  cs := keep_copy;
          bs := false;  er := false;
          end;
        end
      else (* not bus_hit *)
        begin
        bus_release_mutex;
        with bus_status do
          begin
          sl := false; cs := false;
          bs := false; er := false;
          end
        end;
      assign bus_status to st;
      assign Lo to ak_tx;  assign Z to ai_tx;
      end;
    end;  (* do_address_beat *)
```

Figure 6. The address beat subprocess.

```
subprocess do_data_beat :
  upon (connection_status in [selected, reflecting, intervening])
    and ( ((AS_rx = Lo) and (DS_rx = Lo)
                          and (recall(DS_rx) = Hi)) (* odd beat *)
          or ((AS_rx = Lo) and (DS_rx = Hi)
                            and (recall(DS_rx) = Lo)) ) (* even or null beat *)
  check DS_rx do
    begin (* do_data_beat *)
    with bus_addr_command do
      begin
      if connection_status = selected then
        begin (* handle broadcast update *)
        ...
        end
      else (* connection_status in [reflecting, intervening] *)
        begin (* handle third party participation *)
        if BT or (DS_rx = Lo) then
          begin (* odd or even beat, not null data beat *)
          bus_data_command := CM;
          read_occurred := read_occurred or not bus_data_command.WR;
          if bus_data_command.WR then
            begin (* accept data from AD using lane disables *)
            ...
            end
          else (* not WR *)
            begin (* apply data to AD using lane disables *)
            ...
            end;
          (* determine status response *)
          with bus_status do
            begin
            if BT then
              if bus_quadlet_index = line_size-1 then
                begin (* end of line, so wrap to beginning *)
                bus_status.ed := true;  bus_quadlet_index := 0;
                end
              else
                begin (* increment to next quadlet *)
                bus_status.ed := false;
                bus_quadlet_index := bus_quadlet_index + 1;
                end;
            bus_status.sl := false;  bus_status.er := false;
            bus_status.cs := keep_copy;
            end;
          assign bus_status to st;
          end;
        (* handshake as third party *)
        if DS_rx = Lo then (* odd beat *)
          assign Z to di_tx
        else (* even or null beat *)
          assign Lo to di_tx;
        end;
      end;
  end;  (* do_data_beat *)
```

Figure 7. The data beat subprocess.

Once access is gained, the cache determines whether it has a hit at the address of the bus transaction, by checking the *valid* attribute of the corresponding line. If there is no hit, then no further action is required for the transaction, and so exclusive access to the line is released, no Futurebus status signals are asserted, and the cache remains disconnected.

If the cache does detect a hit, its actions depend on the master's command, the remaining attributes of the line, and the cache's configuration settings. For example, in the case of a broadcast update transaction (*IM* and *BC* command bits set), a properly configured cache can connect as a selected slave to receive the new data. The remainder of the subprocess body sets information structure flags for use by the data beat and end beat subprocesses, and specifies the Futurebus status and handshaking for the address beat.

The subprocess which handles data beats is shown in Figure 7. It is sensitive to changes in the data strobe input (*DS_rx*), but is only activated when the cache is connected, either to accept broadcast data, or to participate as a third party. This latter case is shown in detail. The data beat command, consisting of a write signal (*WR*) and four byte-lane disables (*LW*, . . . *LZ*, c.f. byte-enables on other buses), is accepted from the master and used to control the reading or writing of data. Next the Futurebus status is determined, based on whether or not a line wrap has occurred, and then the data beat handshaking is performed.

The end beat subprocess, shown in Figure 8, specifies how a cache updates its line attributes at the end of a transaction. Again, only the detail for third party participation is shown. Where the cache is allowed to keep a copy of the line and intervened on a transaction initiated by a caching master (indicated by *CC* being true), the cache must include the *shared* attribute in the line's status set. If the cache reflected, and shared memory was completely updated with the modified (dirty) data from the line, then, depending on the cache's configuration, it modifies the line attributes accordingly. If the cache had detected a cache hit, exclusive access to the line is released. The last action of the cache is to complete the transaction handshake.

In order to be able to execute the specification of cache behaviour, it is necessary to create a *driver* symbol and corresponding HHDL model for the client interface and

```
subprocess do_end_beat :
  upon not master and (AS_rx = Hi) and (recall(AS_rx) = Lo)
  check AS_rx do
    begin (* do_end_beat *)
    if bus_hit then
      begin
      bus_disconnect_command := CM;
      with bus_addr_command do
        if connection_status = selected then
          begin (* end of broadcast update *)
          ...
          end
        else if connection_status in [reflecting, intervening] then
          (* end of third party participation *)
          if bus_disconnect_command.DBS then
            begin (* do transaction back-out *)
            ...
            end
          else if bus_disconnect_command.DER then
            begin (* do transaction error recovery *)
            ...
            end
          else (* successful completion of transaction *)
            with store[bus_line_index] do
              if keep_copy and not (CC and IM and not BC and read_occurred) then
                (* allowed to keep the line *)
                if connection_status = intervening then
                  if CC then
                    status := status + [shared]
                else (* connection_status = reflecting *)
                  if all_dirty_reflected then
                    if config.keep_after_reflect then
                      if CC then
                        status := status + [shared];
                      if not config.own_after_reflect then
                        status := status - [owner]
                    else (* not config.keep_after_reflect *)
                      status := []
                  else (* not all_dirty_reflected *)
                    if CC then
                      status := status + [shared]
              else (* must invalidate *)
                status := []
        else (* connection_status = unselected *)
          begin (* end of non-participation *)
          ...
          end;
      bus_release_mutex;
      end; (* if bus_hit *)
    (* do handshake to complete disconnection beat *)
    assign Lo to ai_tx;
    assign Z to dk_tx;  assign Z to di_tx;
    assign Z to ak_tx;
    end;  (* do_end_beat *)
```

Figure 8. The end beat subprocess.

for the Futurebus interface. A circuit is created with nets linking the interface pins of a cache symbol instance to an instance of each of the driver symbols. The circuit is then linked to create a simulator. The purpose of the driver models is to activate the input pins of the cache model according to some predetermined command file or algorithm. They can also monitor the output pins of the cache model and report through the trace write mechanism. Using this approach, the behaviour of the cache under various driving conditions can be observed. For example, the client and Futurebus models could be programmed to stimulate the cache with transactions using the same cache line at the same simulation time point, in order to investigate the effects of collision of mutual exclusion requests.

5. CONCLUSIONS

In this paper, we have shown that an information structure model for describing the behaviour of coherent caches has several advantages over other modelling techniques, particularly where the model is to be used as a reference document. It combines an appropriate degree of precision and completeness, while remaining intelligible to an audience not expert in reading highly formal notations. In addition, a specification based on an information structure model expressed in programming language form can be executed to simulate the system being modelled. This simulation can be used as a reference implementation, and can provide test vectors for conformance validation. Where more formal specification is required, well known-semantic specification techniques for programming languages and data types can be applied.

The model we have described can be used for a number of purposes, all being investigated by the authors. Firstly, the IEEE P896.2 Cache Coherence Task Group is drafting a specification of the Futurebus cache coherence protocol based on the ideas presented here. Secondly, the previously published cache coherence strategies can be described in terms of this model, by defining a set of configuration constants for each strategy. Thirdly, a formal proof can be constructed to show that any cache system conforming to the specification does, in fact, maintain coherence. Fourthly, experiments can be performed to determine how various aspects of conforming coherent caches affect overall system performance. The outcomes of these investigations will add significantly to the understanding of cache behaviour in multiprocessor systems.

6. ACKNOWLEDGEMENTS

The authors are grateful for the helpful comments of one of the referees.

7. REFERENCES

ANZELMO, T., MOORE, R. and BELL, C.G. (1985): "Multiprocessor Makes Parallelism Work", *Electronics*, Vol. 58, No. 35, pp. 46-48, 2 September 1985.

ARCHIBALD, J. and BAER, J-L. (1986): "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", *ACM Transactions on Computer Systems*, Vol. 4, No. 4, pp. 273-298, November 1986.

ASHENDEN, P.J., BARTER, C.J. and MARLIN, C.D. (1987) "The Leopard Workstation Project", *ACM Computer Architecture News*, Vol. 15, No. 4, pp. 40-51, September 1977.

ASHENDEN, P.J. and KNIGHT, D.L. (1985): *L-Bus Specification*, Department of Computer Science, University of Adelaide, South Australia.

BASILI, V.R. (1975): "A Structured Approach to Language Design", *Computer Languages*, Vol. 1, No. 3, pp. 255-273, September 1975.

DIGITAL EQUIPMENT CORPORATION (1985): *VAXBI Technical Summary*, Maynard, Massachusetts.

FIELLAND, G. and RODGERS, D. (1984): "32-bit Computer System Shares Load Equally Among up to 12 Processors", *Electronic Design*, pp. 153-168, 6 September 1984.

FREIDEL, D., MARLIN, C.D. and OUDSHOORN, M. "Modelling Communication in Ada with Shared Data Abstractions", in preparation.

GOGUEN, J.A. (1975): "Correctness and Equivalence of Data Types", *Mathematical Systems Theory, Proc. Initial Symposium*, Springer-Verlag, pp. 352-358.

GOGUEN, J.A., THATCHER, J.W., WAGNER, E.G. and WRIGHT, J.B. (1977): "Initial Algebra Semantics and Continuous Algebras", *J. ACM*, Vol. 24, No. 1, pp. 68-95.

GOODMAN, J. (1983): "Using Cache Memory to Reduce Processor-Memory Traffic", *Proc. 10th Ann. Int. Symp. on Computer Architecture*, Stockholm, pp. 124-131, June 1983.

GUTTAG, J.V. (1980): "Notes on type abstraction (Version 2)", *IEEE Transactions on Software Engineering*, Vol. SE-6, pp. 13-23, January 1980.

GUTTAG, J.V., HOROWITZ, E. and MUSSER, D.R. (1978): "The Design of Type Abstractions" in R.T. Yeh, editor, *Current Trends in Programming Methodology*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, Ch. 4, pp. 60-79.
HILL, M. *et al.* (1986): "Design Decisions in SPUR", *IEEE Computer*, Vol. 19, No. 11, pp. 8-22, November 1986.
IEEE Standard 896.1 (1987): *Backplane Bus Specification for Multiprocessor Architectures (Futurebus)*, IEEE, New York, NY.
IEEE Draft Standard 896.2, *Firmware Protocols for Futurebus*, IEEE, New York, NY, forthcoming.
IEEE Draft Standard 1014 (1987): *"VMEbus", A Standard Specification for a Versatile Backplane Bus*, IEEE, New York, NY.
IEEE Standard 1296 (1987): *"High Performance 32-bit Bus*, IEEE, New York, NY.
ISO/TC97/SC21 DP-8807 (1985): *LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, Paris, February 1985.
KATZ, R.H. *et al.* (1985): "Implementing a Cache Consistency Protocol", *Proc. 12th Ann. Int. Symp. on Computer Architecture*, Boston, Masachusetts, pp. 276-283, June 1985.
MANUEL, T. (1987): "How Sequent's New Model Outruns Most Mainframes", *Electronics*, Vol. 60, No. 11, pp. 76-78, 28 May 1987.
MARLIN, C.D. and OUDSHOORN, M. (1985): "Using Abstract Data Types in a Model of the Data Control Aspects of Programming Languages", *Australian Comp. Sci. Commun.*, Vol. 7, No. 1, pp. 19-1—19-10, February 1985.
MOSZKOWSKI, B. "Executing Temporal Logic Programs", *Seminar on Concurrency* (Brookes, S.D. *et al.*, ed.), Lecture Notes in Comp. Sci., No. 197, Springer-Verlag, Berlin, pp. 111-130.
OUDSHOORN, M. and MARLIN, C.D. "Describing Data Control in Programming Languages", in preparation.
PAPAMARCOS, M.S. and PATEL, J.H. (1984): "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories", *Proc. 11th Ann Int. Symp. on Computer Architecture*, Ann Arbor, Michigan, pp. 348-354, June 1984.
PETERSON, J.L. (1977): "Petri Nets", *ACM Computing Surveys*, Vol. 9, No. 3, pp. 223-252, September 1977.
SILVAR-LISCO CORPORATION (1986): *Helix Reference Manual*, Document Nos. HLX-2.2-002, HLX-2.2-003, HLX-2.2-004, Menlo Park, California, October 1986.
SMITH, A.J. (1982): "Cache Memories", *ACM Computing Surveys*, Vol. 14, No. 3, pp. 473-530, September 1982.
SWEAZEY, P. and SMITH, A.J. (1986): "A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus", *Proc. 13th Ann. Int. Symp. on Computer Architecture*, Tokyo, Japan, pp. 414-423, June 1986.
TENNENT, R.D. (1976): "The Denotational Semantics of Programming Languages", *Commun. ACM*, Vol. 19, No. 8, pp. 437-453, August 1976.
VERNON, M.K. and HOLLIDAY, M.A. (1985): *Performance Analysis of Multiprocessor Cache Consistency Protocols Using Generalized Timed Petri Nets*, Tech. Report, Computer Sciences Department, University of Wisconsin, Madison.
WEGNER, P. (1971): "Data Structure Models for Programming Languages", *Proc. Symp. Data Structures in Programming Languages*, pp. 1-54.

BIOGRAPHICAL NOTES

Peter Ashenden is a Senior Research Officer with the Department of Computer Science at the University of Adelaide. He completed his Honours Degree in Computer Science in 1982 at the University of Adelaide and continued there as a Research Assistant when the Leopard Project was founded. His main areas of research interest are computer architecture and computer engineering, particularly related to multiprocessor architectures. He is also interested in computer-aided engineering for electronics design.

Chris Marlin has been with the Department of Computer Science at the University of Adelaide, where he is now a Senior Lecturer and Deputy Chairman, since 1984. He completed his Honours Degree in Computing Science in 1973 and his PhD in Computing Science in 1979, both at the University of Adelaide. From January 1980 to December 1983, he was an Assistant Professor of Computer Science at the University of Iowa, Iowa City, Iowa (USA). His research has primarily been concerned with programming language design, specification and implementation, especially in relation to coroutines and parallel processes, and various aspects of integrated incremental programming environments.

References

[1] R. A. Altmann, A. N. Hawke and C. D. Marlin, “An integrated programming environment based on multiple concurrent views,” *Australian Computer Journal*, Vol. 20, No. 2 (May 1988), pp. 65–72.

[2] J. Archibald and J.-L. Baer, “Cache coherence protocols: evaluation using a multiprocessor simulation model,” *ACM Transactions on Computer Systems*, Vol. 4, No. 4 (November 1986), pp. 273–298.

[3] P. J. Ashenden and C. D. Marlin, “A behavioural specification of cache coherence,” *Australian Computer Journal*, Vol. 20, No. 2 (May 1988), pp. 50–57.

[4] P. J. Ashenden, *L-Bus specification, Version 1.3*, Department of Computer Science, University of Adelaide, South Australia (March 1986).

[5] P. J. Ashenden, C. J. Barter and M. A. Petty, *The Leopard multiprocessor workstation project*, CCSSE Tech. Report LW-01 (November 1989), Dept. Computer Science, University of Adelaide, South Australia.

[6] P. J. Ashenden, *The Leopard-2 workstation bus architecture*, CCSSE Tech. Report LW-02 (August 1989), Dept. Computer Science, University of Adelaide, South Australia.

[7] P. J. Ashenden, R. Gerhofer and K. R. Howard, *The Leopard-2 General Data Processor users guide*, CCSSE Tech. Report LW-03 (September 1989), Dept. Computer Science, University of Adelaide, South Australia.

[8] P. J. Ashenden, C. Fang, R. Gerhofer, K. R. Howard and G. C. Slater, *The Leopard-2 General Data Processor design description*, CCSSE Tech. Report LW-04 (March 1990), Dept. Computer Science, University of Adelaide, South Australia.

[9] P. J. Ashenden, *The Leopard-2 Futurebus Monitor users guide*, CCSSE Tech. Report LW-05 (September 1989), Dept. Computer Science, University of Adelaide, South Australia.

[10] P. J. Ashenden, *The Leopard-2 Futurebus Monitor design description*, CCSSE Tech. Report LW-06 (March 1990), Dept. Computer Science, University of Adelaide, South Australia.

[11] P. J. Ashenden, *The Leopard-2 General Data Processor local bus description*, CCSSE Tech. Report LW-07 (September 1989), Dept. Computer Science, University of Adelaide, South Australia.

[12] P. J. Ashenden, *The Leopard-2 Futurebus Interface functional description*, CCSSE Tech. Report LW-10 (November 1989), Dept. Computer Science, University of Adelaide, South Australia.

[13] P. J. Ashenden, *The Leopard-2 General Data Processor local memory design description*, CCSSE Tech. Report LW-12 (March 1990), Dept. Computer Science, University of Adelaide, South Australia.

[14] P. J. Ashenden, *The Designer's Guide to VHDL*, Morgan Kaufmann Publishers , Inc., San Francisco (1996).

[15] L. M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Transactions on Computers*, Vol. C-27, No. 12 (December 1978), pp. 1112–1118.

[16] Digital Equipment Corporation, *Alpha 21164 Hardware Reference Manual,* http://ftp.digital.com/pub/Digital/info/semiconductor/literature/164hrm.pdf (1997).

[17] M. Dubois and F. A. Briggs, "Effects of Cache Coherency in Multiprocessors," *IEEE Transactions on Computers*, Vol. C-31, No. 11 (November 1982), pp. 1083–1099.

[18] S. J. Eggers and R. H. Katz, "A characterization of sharing in parallel programs and its application to coherency protocol evaluation", Proc. 15th Int. Symp. Computer Architecture, *ACM Computer Architecture News*, Vol. 16, No. 2 (June 1988), pp. 373–382.

[19] C. Fang, *A high performance colour graphics display system*, Masters Thesis (November 1987), Department of Computer Science, The University of Adelaide, South Australia.

[20] M. J. Flynn, "Very high speed computing systems," *Proceedings of the IEEE*, Vol. 54, No. 12 (December 1966), pp. 1901–1909.

[21] S. J. Frank, "Tightly coupled multiprocessor system speeds memory-access times," *Electronics*, Vol. 57, No. 1 (January 1984), pp. 164–169.

[22] H. Garsden and A. L. Wendelborn, "A comparison of microtasking implementations of the applicative language SISAL," in H. Burkhart (ed.), *Proc. COMPAR 90–VAPP IV (Joint International Conference on Vector and Parallel Processing Switzerland)*, Lecture Notes in Computer Science, Vol. 457, Springer, Berlin (1990).

[23] R. Gerhofer, C. Fang and P. J. Ashenden, *The Leopard-2 Storage and Communications Processor local bus description*, CCSSE Tech. Report LW-19 (April 1990), Dept. Computer Science, University of Adelaide, South Australia.

[24] N. D. Godiwala and Barry A. Maskas, "The Second-generation Processor Module for AlphaServer 2100 Systems," *Digital Technical Journal,* Vol. 7, No. 1, http://www.digital.com/info/DTJH06/DTJH06SC.TXT (1995).

[25] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," Proc. 10th Int. Symp. Computer Architecture, *ACM Computer Architecture News*, Vol. 11, No. 3 (June 1983), pp. 124–131.

[26] J. L. Hennessy and D. A. Patterson, *Computer Architecture: a Quantitative Approach*, 2nd Edition, Morgan Kaufmann Publishers, San Mateo, Calif. (1996).

[27] M. Hill *et al.*, "Design decisions in SPUR", *IEEE Computer*, Vol. 19, No. 11 (November 1986), pp. 8–22.

[28] M. A. Holliday and M. K. Vernon, "A Generalized Timed Petri Net model for performance analysis," *IEEE Trasactions on Computers*, Vol. SE-13, No. 12 (December 1987), pp. 1297–1310.

[29] C. Hunter, *Series 32000 programmer's reference manual*, Prentice-Hall, New Jersey (1987).

[30] IBM, *PowerPC 604 RISC Microprocessor User's Manual,* http://www.chips.ibm.com/products/ppc/documents/datasheets/604/user_manual/604um.pdf (1995).

[31] IEEE, *IEEE standard backplane bus specification for multiprocessor architectures: Futurebus*, ANSI/IEEE Std. 896.1-1987, IEEE, New York (1988).

[32] IEEE P896.2 Working Group, *Futurebus P896.2 specification*, Draft 1.1 (August 1988), IEEE Inc., New York.

[33] IEEE, *Standard VHDL Language Reference Manual*, IEEE Std. 1076–1993, IEEE, New York (1993).

[34] Intel Corp., *Intel Pentium® Processor Family Developer's Manual*, http://developer.intel.com/design/pentium/manuals/24142805.pdf (1997).

[35] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins and R. G. Sheldon, "Implementing a cache consistency protocol," Proc. 12th Int. Symp. Computer Architecture, *ACM Computer Architecture News*, Vol. 13, No. 3 (June 1985), pp. 276–283.

[36] D. L. Knight, P. J. Ashenden, C. D. Marlin and C. J. Barter, "The QDS-1000: a modular expandable image processing workstation", presented at *Remote Sens-*

ing—Current Status and Applications, South Australian Institute of Technology, Adelaide, South Australia (June 1985).

[37] E. McCreight, *The dragon computer system: an early overview*, Technical Report, Xerox Corp. (September 1984), cited by Archibald and Baer in [2].

[38] MIPS Technologies, Inc., *R4000 / R4400 Microprocessor User's Manual, 2nd edition*, ftp://sgigate.sgi.com/pub/doc/R4400/User_Manual/
R4400_Uman_book_Ed2.pdf (1994).

[39] M. S. Papamarcos and J. H. Patel, "A low-overhad coherence solution for multiprocessors with private cache memories", Proc. 11th Int. Symp. Computer Architecture, *ACM Computer Architecture News*, Vol. 12, No. 3 (June 1984), pp. 348–354.

[40] J. H. Patel, "Analysis of multiprocessors with private cache memories", *IEEE Transactions on Computers*, Vol. C-31, No. 4 (April 1982), pp. 296–304.

[41] F. Pong and M. Dubois, *The verification of cache coherence protocols*, Technical Report No. CENG-92-20 (November 1992), Dept. Electrical Engineering—Systems, University of Southern California, Los Angeles.

[42] P. Robinson, *The IEEE Futurebus cache coherence protocol as a logic program*, unpublished memo (1988), Computer Laboratory, University of Cambridge.

[43] M. Rozier *et al.*, *CHORUS distributed operating system*, Chorus Systèmes Tech. Report CS/TR-88-7.6 (November 1988).

[44] L. Rudolph and Z. Segall, "Dynamic decentralized cache schemes for MIMD parallel processors", Proc. 11th Int. Symp. Computer Architecture, *ACM Computer Architecture News*, Vol. 12, No. 3 (June 1984), pp. 340–347.

[45] C. Scheurich and M. Dubois, "Correct memory operation of cache-based multiprocessors," Proc. 14th Int. Symp. Computer Architecture, *ACM Computer Architecture News*, Vol. 15, No. 2 (June 1987), pp. 234–243.

[46] Silvar-Lisco Corp., Helix reference manual, Document Nos. HLX-2.2-002, HLX-2.2-003 and HLX-2.2-004, Menlo Park, CA (October 1986).

[47] Sun Microsystems, Inc., *SPARCsystem™ 600MP: new technology for flexibility, scalability, and growth*, Technical White Paper (September 1991).

[48] Sun Microsystems, Inc., *UltraSPARCTM-II High–Performance, 250 MHz, 64–Bit RISC Processor,* http://www.sun.com/sparc/stp1031/datasheets/stp1031lga.pdf (1997).

[49] I. E. Sutherland, C. E. Molnar, R. F. Sproull and J. C. Mudge, "The TRIMOS-BUS," *CalTech Conference on VLSI* (January 1979), pp. 395–427.

[50] P. Sweazey and A. J. Smith, "A class of compatible cache consistency protocols and their support by the IEEE Futurebus," Proc. 13th Int. Symp. Computer Architecture, *ACM Computer Architecture News*, Vol. 14, No. 2 (June 1986), pp. 414–423.

[51] A. S. Tanenbaum, *Operating systems: design and implementation*, (Prentice Hall, 1987).

[52] C. Thacker, L. C. Stewart and E. H. Satterthwaite, "Firefly: a multiprocessor workstation", *IEEE Transactions of Computers*, Vol. 37, No. 8 (August 1988), pp. 909–920.

[53] M. K. Vernon and M. A. Holliday, "Performance analysis of multiprocessor cache consistency protocols using Generalized Timed Petri Nets", *Proc. Performance 86 and ACM SIGMETRICS 1986 Joint Conference on Computer Performance Modeling, Measurement and Evaluation*, Raleigh, N.C. (May 1986), pp. 9–17.

[54] M. K. Vernon, E. D. Lazowska and J. Zahorjan, "An accurate and efficient performance analysis technique for multiprocessor snooping cache-consistency protocols", Proc. 15th Int. Symp. Computer Architecture, *ACM Computer Architecture News*, Vol. 16, No. 2 (June 1988), pp. 308–315.

[55] D. A. Wood, G. A. Gibson and R. H. Katz, "Verifying a multiprocessor cache controller using random test generation," *IEEE Design and Test of Computers*, Vol. 7, No. 4 (August 1990), pp. 13–25.