CACHE IMPLEMENTATION FOR MULTIPLE MICROPROCESSORS

C. V. Ravishankar
James R. Goodman

Department of Computer Sciences
University of Wisconsin-Madison
Madison, WI 53706

ABSTRACT

This paper presents an organization for a cache memory system for use in a microprocessor-based system structured around the Multibus* or some similar bus. Standard dynamic random access memory (DRAM) is used to store the data in the cache. Information necessary for the control of and access to the cache is held in a specially designed VLSI chip. The feasibility of this approach has been demonstrated by designing and fabricating the VLSI chip and a test facility. The critical parameters and implementation details are discussed.

This implementation supports multiple cards, each containing a processor and a cache, as described in [Goodman83]. The technique involves monitoring the bus for references to main storage. The contention for cache cycles between the processor and the bus is resolved by using two identical copies of the tag memory.

Introduction

Microprocessor systems today are frequently built using a standard bus structure to connect the printed circuit boards. Of these standard buses, by far the most popular is the Multibus or some variation, such as the proposed IEEE P796 standard. The typical configuration of a processor card in such a system includes, along with the processor, an amount of memory -- typically up to 16K bytes of ROM and 32K bytes of RAM. Usually the RAM may be accessed by other bus masters through the bus, and so is additional memory contained in memory boards. However, the access rate over the bus is so low that even a single processor suffers a severe speed penalty unless it accesses its local memory nearly all the time.

This organization has been shown to be effective in some environments: where for example, each processor is pre-assigned a fixed task. However, it has proved unsatisfactory in others, in particular when several processors attempt to share a common set of tasks. In such cases, it has been difficult to partition the set of tasks so that the number of accesses over the bus is kept low.

One way around this problem is to replace the local memory with cache memory. Caches, which have long been used on high performance computers [Liptay68], have been designed into many minicomputers[Bell78, DataG80], and are now starting to appear in microprocessor systems [Isaak82]. They work by taking advantage of the spatial and temporal locality observed in memory references patterns, and even with a small cache, 90% or more of accesses are typically to data held in the cache.

When main storage has to be accessed, a cache exploits this spatial locality by fetching a block of words surrounding the accessed word, as neighboring locations are most likely to be referenced next. It exploits temporal locality in the way that it replaces entries in the cache -- it aims at throwing out the items that will not be referenced for the longest time hence.

However, fetching large blocks of data into the cache each time results in highly undesirable surges in the transfer rate across the bus. It has recently been shown [Goodman83] that even if only temporal locality is primarily present but task switching is relatively infrequent, a cache can remain very effective by fetching very small blocks. We believe that this technique is applicable to microprocessor environments like the one above, provided we have a sufficient number of processor-cache pairs so that each processor may be assigned a single task. This approach, however, introduces a new problem, known as the *stale data* problem.

Maintaining Cache Consistency

The stale data problem in cache schemes for multiprocessors arises since multiple copies of a datum from main storage may be distributed among the individual caches, [Censier78, Tang76] but the appearance of exclusive access to main storage must be maintained at the level of each processor. While the problem is well-understood, solutions tend to be messy, involving either invalidation requests to all the caches every time any datum is modified, or a complex bookkeeping structure which maintains information such as what data is in the caches; and if the current copy of a datum is not in main storage, which cache it is in.

Recently a clean, consistent solution was proposed for any system containing a common bus for memory accesses [Goodman83]. This scheme, to be described presently, makes use of normal *read* and *write* signals to maintain consistency.

It has the feature that while consistency is maintained by all the bus masters acting in concert, the operation of each of the masters is independent of the others. In particular, any bus master depends only on the knowledge of the contents of its own cache.

This technique has the advantages of both the conventional techniques for maintaining compatibility between the cache and the main storage: write-through, (where main storage is promptly updated upon modification of a datum in any cache), and write-back

*Multibus is trademark of Intel Corporation.

CH1856-4/83/0000-0346$1.00 © 1983 IEEE

(where a block is not written out to main storage until the replacement algorithm decides to purge the block from the cache).

This paper presents an implementation of a solution that requires a write through only at the first modification of a datum in a cache -- a scheme that we call *write-once*. We have determined that such a scheme can be implemented using the standard Multibus.

The Write-Once Control Scheme

The current implementation is structured around the Intel "Multibus" as the communication-arbitration unit: the Multibus has gained wide acceptance, and the new IEEE standard P796 is based on the Intel design. Many manufacturers already offer Multibus-compatible cards. The Multibus also has a feature that is crucial to our design: it has an inhibit line that may be raised by any device on the bus to inhibit accesses from reaching the main store. As explained below, this feature ensures that we are able to maintain the appearance of exclusive access to main storage from the perspective of each processor. The operation of the scheme is as follows:

(1) Each processor accesses its cache for reads and writes. If the data is present in the cache, the operation completes successfully. If the data is not resident in the cache, a request is put out on the bus to fetch the data.

(2) Whenever a word in a cache is modified, the cache controller controlling the cache sets a flag (corresponding to the "dirty" bit used in many other implementations) to indicate this fact. Further, a write to main storage is initiated immediately upon the first modification of the datum: the associated bus controller partition grabs the bus, and writes through to main storage. Since all the other bus controller partitions monitor the bus, they all now can test if their cache holds a copy of the datum being written out; the write to main storage serves as the signal to them to invalidate any such copies. We therefore need no special "invalidate" signals.

(3) The response to read requests on the bus is a little different: whenever a partition records a request on the bus for a datum of which its cache possesses a valid copy, the partition immediately inhibits the request from reaching main store, and responds to the request by placing on the bus the valid copy it obtains from its cache. Hence, the copy in main storage need be updated only upon the first modification -- all subsequent requests are inhibited from reaching main storage, and are handled by the preempting controller partition.

Information regarding the contents of the cache is held as usual in an address array. In our implementation, the address array also holds information regarding whether a datum has been modified or not and the LRU information used in the replacement algorithm.

Since the bus controller and the processor are both constantly active, and need access to the information in the address array, the potentially severe contention between them is resolved by using two address arrays holding identical tables: one for the exclusive use of the processor - the other for the bus controller. At the heart of the current implementation is a single VLSI chip that we have designed (in nMOS technology) that contains the address array and some of it's associated logic.

Overview of the Cache Design

In this paper we describe a reasonable cache organization for a single board processor intended for use in a Multibus system. Refer to Fig. 1.

tag field	R field	B field

Fig. 1. Breakdown of Address Word for Cache Organization.

The address supplied by the processor is broken down into three fields:

B This field, consisting of b bits, specifies which word is to be selected from within a block.

R This field, consisting of r bits, specifies the row in which the addressed block resides.

TAG This field, consisting of t bits, is the portion of the address which is saved in the address array to uniquely identify the main memory location currently resident in the cache block.

A note about the use of the term *word:* in this discussion we mean the amount of data written into the cache with one bus access, i.e., the width of the bus. For Multibus, this is 16 bits, but nothing in our implementation prevents the use of a wider bus. A larger word could also be assembled by a sequence of transfers of data across the bus. A word also is normally the amount of data available to the processor from the cache in a single access, though in fact only part of the data may actually be transmitted to the CPU. Note that this usage may vary from the normal meaning of a word and, in particular, means that we are ignoring in the description of the address those bits which specify a finer granularity of access.

Fig. 2 shows the cache model. The organization contains r rows and is known as n-way set-associative. It consists of n identical columns, each consisting of two segments: the data memory and the tag memory. In addition, the cache includes an LRU segment containing information used by the replacement algorithm for determining the column which contains the block to be purged when a new block is fetched. This segment contains r rows of $p = \frac{n(n-1)}{2}$ bits each to indicate an LRU ordering among the n columns.

The data memory contains the data corresponding to a given location in main memory. Each block of 2^b words in main memory can be stored in any of the n places of the appropriate row. The tag memory contains a tag corresponding to each block data identifying the corresponding main memory block whence this data came. In addition, it contains some other information:

(1) $v = 2^b$ pairs of *mode* bits indicating which words of memory are present (some may not have been fetched and are therefore empty).

(2) 2 bits to indicate the validity of the data.

Each block from main memory may be stored in any of the n places in the appropriate row.

When a request is received from the processor, the R-field is used as a row index into the tag and data memories simultaneously. Each of the tag fields read out (from the tag memory) for the appropriate row is compared against the tag field of the address. If a match occurs, and the corresponding mode field indicates valid data, a *hit* is signaled, and the cache supplies the data to the processor. If no match is found, or if the mode field indicates that the word is not present, then a miss is indicated, and a main memory operation is initiated. During this time a block is chosen as victim to be purged to make way for the new block. This choice is

made on the basis of the p-bits which may be interpreted as a list of the set elements in order of their reference. The least recently referenced block is chosen.

The cache can be implemented with conventional dynamic random access memory for the data part, and a special VLSI chip for the tag part. This solution is particularly appropriate for VLSI; while it allows the use of the cheapest form of semiconductor memory available -- dynamic RAM organized as one bit per chip, it nicely partitions the problem into pieces which have minimal intercommunication requirements: only the results of the comparisons need to be reported off the chip, and this requires about $lg_2 n$ pins.

At each access, at most one word of the data need be read out from the data memory, and a multiplexing of the multiple outputs for this purpose can actually be accomplished inside the dynamic RAM by selecting the appropriate set through addressing. If the VLSI part implementing the tag array is fast enough to be able to supply the addressing bits in time for the column decode within the RAM, the data can still be read at the maximum rate of the dynamic RAM.

Cache Design Parameters

To demonstrate the feasibility of this approach, we have designed and submitted for fabrication a circuit which implements a major piece of the tag memory in VLSI using nMOS technology. Because of the limitations of the technology available to us, we found it necessary to partition the part into multiple (but identical) chips, but it does seem that a state-of-the-art implementation could produce the entire tag memory on a single chip.

In choosing the parameters for the implementation, we recognized that we were demonstrating the feasibility of an architecture rather than designing a commercially viable product. We planned to use the Mead-Conway [Mead80] approach to build a chip which consisted largely of memory. For memory intensive designs, the Mead-Conway tools may not be particularly effective. Enormous effort is usually expended in optimizing the design of a single memory cell because of its great impact on the ultimate size of the chip. We thus recognized that, while we might end up with a large chip, it would contain far less than that possible with state-of-the-art commercial design capability. Therefore we partitioned the chip in a way that we could design only one, but use several to accomplish what is commercially feasible on a single chip. This was done by partitioning the chip into columns, with each chip containing the tag memory for one column plus two bits to implement the LRU segment. This allows us to build a cache up to 4-way set associative.

We picked 64 rows (r=6) arbitrarily, since our studies have indicated that this is both a reasonable and typical number of rows. The tag field we used was large, perhaps larger than appropriate, but we wanted to be able to evaluate our system with one of several possible processors, and the tag field is not readily expandable.

At the time we initiated the design, we had not evaluated the effectiveness of the partial block retrieval, so we assumed one tag for each word of memory (i.e., v = 1, or b = 0). Since that time, studies reported in [Goodman83] have indicated that the optimal value for v is probably eight or sixteen.

Since we began this work, a commercial product with apparently similar goals has been announced [TI82]. This chip, known as the TMS 2150 Cache Address Comparator, has four times as much on-chip memory as ours. The designers of the TI part also chose v = 1. Rather than being 4-way set associative, as we suggest, it is implemented as one very tall column (512 rows). This means that constructing a 2-way set-associative cache requires a minimum of two chips, providing for a minimum of 1024 blocks. Based on our simulation studies, we believe that our organization is more cost-effective, particularly for an implementation where $v > 1$. The TI design also apparently is not intended for a multiprocessor environment.

Implemented Solution

As explained above, the cache as planned was to be 4-way set associative, and the implementation was to be in VLSI. Since it seemed unlikely that it would be possible to house the entire amount of storage on a single chip, an early decision was made to partition the controller so that each chip as visualized held only one of the four address arrays. Such a vertical partition has seemed the most reasonable approach.

The current implementation therefore uses four identical VLSI chips that essentially house the address arrays and some of the logic. It may be remarked that it is easy to implement an n-way Set-associative cache using n such chips supported by some external logic. This was part of the motivation for the mode of partition. The address array uses no clock and is essentially read-through in design: it operates in the "read" mode so long as the "write" line is low. This was done since it is reasonable to expect most of the accesses to be reads, and so for reasons of speed the design aimed at having the data on reads available essentially after the address lines stabilized. A static RAM was preferred for the address array storage to ensure faster response, and because the chip as designed uses no clock. Since the implementation was in nMOS, a pre-charge signal was specified in order that the accesses be speeded up and so that no complicated differential sensing schemes need be used in this preliminary version to control bus precharging for the dual ported memory cells.

Implementation Details

Most of the area on the chip is occupied by the storage and the decoders. The basic storage cell is dual ported, and hence the complement of the data bit is also available. The address size is 24 bits, which is partitioned into a 6 bit index into the address array, and the remaining 18 bits are compared with the corresponding contents of the address array to signal a hit or miss. A decoder is used at both ends of the address select lines to speed up the response of the memory.

The pins are as below:

Input pins	
Address lines	24
Precharge	1
Read/write from proc.	1
Clear valid bit	1
Mark & Write signal	1
Load into Address Array signal	1
LRU inputs	2
Output pins	
Hit signal	1
Write through signal	1
LRU outputs	2
Dirty bit	1
Valid bit	1
VDD & GND	2
Total	39

Modifications are being currently made to the output signals from the chip to indicate write-back when data has been modified more than once (refer below for details). This also would alter the pin count slightly when implemented.

The functioning of the chip is as follows: Note that since the chip does not use a clock, it relies on the precharge signal and assumes that address data can be input onto the pads at the same time. The chip also depends upon the cache controller for some of the externally driven functions.

a) The "read" mode:

   1) Precharge is raised and address placed on 24 address pins.

   2) A hit/miss is signaled according as data is present/absent in cache and the following data is output: (i) LRU (ii) dirty bit (iii) valid bit

b) The "write" mode:

   1) The precharge is raised and address placed on 24 address pins.

   2) The read/write signal is raised to indicate "write."

   3) The chip signals a miss if the datum is not present. If it is present then a write-through is signaled if this is the first modification of the datum. The other outputs are the same as in the case of a read.

The *mark & write* signal deserves special comment. The reason for this signal is that the LRU information is input from the controller, and in the time taken to compute the new LRU the processor may have placed a new address value on the input pins thereby selecting a different row of the storage array. If a write is performed now to load the LRU, the information will clearly be written into the wrong row. To circumvent this possibility, the *mark & write* line is raised by the controller as soon as the row selection is complete in the storage array. This now latches a special "mark cell" that drives the select lines to the LRU corresponding to the row being accessed and cuts off this part of the select line from the rest. The LRU cells now remain selected even though the rest of the cells may not. The LRU may be loaded as long as the *mark & write* signal remains high.

As mentioned, a feature that is currently under implementation is a provision for keeping track of whether a given datum has been modified more than once. If the replacement algorithm wishes at some point to overwrite a cache location with some newly fetched data, a write back to memory is necessary only if the datum has been modified more than once since a write through was performed to main store at the first modification. It is possible to use the four states corresponding to different values of the dirty and valid bits to represent the conditions invalid, valid, written once and written more than once. A "write back" is signaled if the datum has been modified more than once. The current version of the chip has a finite state machine on it to perform this (decoding) function.

The chip is currently being fabricated and we expect to have it available for testing soon. Testability was a concern during the design of the chip, and while there isn't a great deal of on-chip logic specifically intended to help during testing, the testing scheme for this chip is fairly straight forward. Some additional off-chip logic would be required, but since all such logic is expected to be in TTL, testing and debugging is not an overwhelming concern.

Summary

We have presented a cache memory system for use in a microprocessor-based system organized around the Multibus or another similar bus. The technique involves monitoring the bus for references to main storage. We presented the block diagram for the cache, which resides on the same Multibus card with the processor. The cache comprises a specially-designed VLSI chip to store the necessary information for control of and access to the cache, and standard dynamic random access memory (DRAM) to store the data in the cache. Multiple processors can be supported as described in [Goodman83] by this implementation. The competition for cache cycles between the processor and the bus is resolved by using two copies of the tag memory.

The approach has been followed up by designing and fabricating a VLSI chip and test bed. The critical parameters and implementation details were presented.

Acknowledgements: We would like to thank Steve Chan, who was one of the two persons who actually did the design and layout of the VLSI chip, and Randy Katz, whose suggestions and advice were of great help during the design and layout phase.

References

[Bell 78] C. G. Bell, J. C. Mudge, and J. E. McNamara, *Computer Engineering: A DEC view of hardware systems design*, Digital Press, Bedford, Massachusetts, 1978.

[Censier 78] L. M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Trans. on Computers*, Vol. C-27, No. 12, December 1978, pp. 1112-1118.

[DataG 80] "Eclipse MV/8000 Principles of Operation," Ordering No. 014-000648, Data General Corporation, Westboro, Massachusetts.

[Goodman 83] J. Goodman, "Using Cache to Reduce Processor/Memory Bandwidth," submitted to *10th Symp. on Computer Arch.*, (June 1983).

[Issak 82] J. Isaak, "Squeezing the most out of the 68000," *Mini-Micro Systems*, Vol. 15, No. 10,

(October 1982), pp. 193-202.

[Liptay 68] J. S. Liptay, "Structural aspects of the System/360 Model 85, II.- The cache", *IBM Systems Journal*, Vol. 7, No. 1, (January 1968), pp. 15-21.

[Mead 80] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[Tang 76] C. K. Tang, "Cache system design in the tightly coupled multiprocessor system," *AFIPS Proc., NCC*, Vol. 45, pp. 749-753, 1976.

[TI 82] *Texas Instruments MOS Memory Data Book*, Texas Instruments, Inc., MOS Memory Division, Houston, Texas, pp. 106-111, 1982.

Fig. 2. Details of Cache Memory Structure.