LOCKUP-FREE INSTRUCTION FETCH/PREFETCH CACHE ORGANIZATION

DAVID KROFT

Control Data Canada, Ltd.
Canadian Development Division
Mississauga, Ontario, Canada

ABSTRACT

In the past decade, there has been much literature describing various cache organizations that exploit general programming idiosyncrasies to obtain maximum hit rate (the probability that a requested datum is now resident in the cache). Little, if any, has been presented to exploit: (1) the inherent dual input nature of the cache and (2) the many-datum reference type central processor instructions.

No matter how high the cache hit rate is, a cache miss may impose a penalty on subsequent cache references. This penalty is the necessity of waiting until the missed requested datum is received from central memory and, possibly, for cache update. For the two cases above, the cache references following a miss do not require the information of the datum not resident in the cache, and are therefore penalized in this fashion.

In this paper, a cache organization is presented that essentially eliminates this penalty. This cache organizational feature has been incorporated in a cache/memory interface subsystem design, and the design has been implemented and prototyped. An existing simple instruction set machine has verified the advantage of this feature; future, more extensive and sophisticated instruction set machines may obviously take more advantage. Prior to prototyping, simulations verified the advantage.

INTRODUCTION

A cache buffer[1,2] is a small, fast memory holding most recently accessed data and its surrounding neighbors. Because the access time of this buffer is usually an order of magnitude greater than main or central memory, and the standard software practice is to localize data, the effective memory access time is considerably reduced when a cache buffer is included. The cost increment for this when compared with the cost of central memory along with the above access time advantage infers cost effectiveness.

Now, accepting the usefulness of a cache buffer, one looks into ways of increasing its effectiveness; that is, further decreasing the effective memory access time. Considerable research has been done to fine tune a cache design for various requirements.[3,6] This fine tuning consisted of selecting optimal total cache buffer size, block size (the number of bytes to be requested on a cache miss), space allocation, and replacement algorithms to maximize hit rate. Another method presented to increase the hit rate was selective prefetching.[7] All these methods assume the cache can handle *only* one request at a time; on a miss, the cache stays busy servicing the request until the data is received from memory and, possibly, for cache buffer update.

In this paper, a cache organization is presented that increases the effectiveness of a normal cache inclusion by using the inherent dual input nature of an overall cache and the many data reference instructions. In other words, it would be extremely useful to pipeline the requests into the cache at the cache hit throughput rate regardless of any misses. If this could be accomplished, then all fetch and/or prefetch of instructions could be totally transparent to the execution unit. Also, for instructions that require a number of data references, the requests could be almost entirely overlapped. Obviously, requests could not be streamed into the cache at the hit throughput rate indefinitely. There is a limit. This organization's limit is imposed by the number of misses that have not been completely processed that the cache will keep track of simultaneously without lockup.

ORGANIZATION

In addition to the standard blocks, this cache organization requires the following:

1. One unresolved miss information/status holding register (MSHR) for each miss that will be handled concurrently.

2. One *n* way comparator, in which *n* is the number of MSHR registers, for registering hits on data in transit from memory.

3. An input stack to hold the total number of received data words possibly outstanding . The size of this stack, consequently, is equal to the

0149-7111/81/0000/0081$00.75 © 1981 IEEE

block size in words times the number of MSHR registers.

4. MSHR status update and collecting networks.

5. The appropriate control unit enhancement to accommodate 1 through 4.

Figure 1 is a simplified block diagram of the cache organization. (A set-associative operation is assumed.) Included are the required blocks for a set-associative cache (tag arrays and control, cache buffer), the central memory interface blocks (memory requestor, memory receiver), and the cache enhancement blocks (miss info holding registers, miss comparator and status collection, input stack). The miss info holding registers hold all necessary information to (1) handle the central memory received data properly and (2) inform the main cache control, through the miss comparator and status collector, of all hit and other status of data in transit from memory. The input stack is necessary to leave the main cache buffer available for overlapped reads and writes. Note that this organization allows for data just received from memory or in the input stack to be sent immediately to the requesting CPU units.

Of course, the number of MSHR registers is important. As with set size (blocks per set), the incremental value decreases rapidly with the number of registers. This is good, because the cost increases significantly with the number of registers. Figure 2 presents a qualitative curve. The average delay time is caused by lockout on outstanding misses. This delay time, of course, is also dependent on cache input request and hit rates. In the degenerate case, 1 MSHR register of reduced size is required; 2 MSHR registers allow for overlap while one miss is outstanding, but still would lock up the cache input on multiple misses outstanding. Owing to cost considerations and incremental effectiveness gained on increasing the number of MSHR registers, 4 registers appear to be optimal.[8]

The necessary information contained within one of these MSHR registers includes the following: First, the cache buffer address, along with the input request address, is required. The cache buffer address is kept to know where to place the returning memory data; the input request address is saved to determine if, on subsequent requests, the data requested is on its way from central memory. Second, input request identification tags, along with the send-to-CPU status, are stored. This information permits the cache to return to CPU requesting units only the data requested and return it with its identification tag. Third, in-input-stack indicators are used to allow for reading data directly from the input stack. Fourth, a code (for example, one bit per byte for partial write) is held for each word to indicate what bytes of the word have been written to the cache buffer. This code controls the cache buffer write update and allows dispensing of data for buffer areas that have been totally written after requested. The cache, thus, has the capability of processing partial write input requests "on the fly" without purging. (Of course, this partial write code may not be incorporated if the cache block is purged on a partial write request to a word in a block in transit from memory.) Last, some control information (the register contains valid information only for returning requested data, but not for cache buffer update and the number of words of the block that have been received and written, if required, into the cache buffer) is needed. Therefore, each MSHR register contains:

1. Cache buffer address
2. Input request address
3. Input identification tags (one per word)
4. Send-to-CPU indicators (one per word)
5. In-input-stack indicators (one per word)
6. Partial write codes (one per word)
7. Number of words of blocks processed
8. Valid information indicator
9. Obsolete indicator (information *not* valid for cache update or MSHR hit on data in transit)

OPERATION

The operation can be split into two basic parts: memory receiver/input stack operations and tag array control operations. For memory receiver/input stack operations, the fields of MSHR interrogated are the following:

1. Send-to-CPU indicator
2. Input identification tags
3. Cache buffer address
4. Partial write codes

5. Obsolete indicator

6. Valid indicator

When a word is received from memory, it is sent to the CPU requesting unit if the send-to-CPU indicator is set; the appropriate identification tag accompanies the data. This word is also written into the input stack if the word's space has not been previously totally written in the cache buffer or if MSHR is not obsolete (invalid for cache update). The words of data are removed from this input stack on a first-in, first-out basis and are written into the cache buffer using fields 3 and 4. Of course, MSHR must hold valid information when interrogated, or an error signal will be generated.

A slight diversion is necessary at this point to explain cache data tagging. On a miss, the cache requests a block of words. Along with each word, a cache tag is sent. This tag points to the particular assigned MSHR and indicates the word of the block. Note that the cache saves in MSHR the requesting unit's identification tag. This tagging closes the remaining open link for the handling of data returned from memory and removes all restrictions on memory on the order of responses.

If a particular processor/memory interface allows for a data width of a block of words for cache to central memory requests, the cache data tagging may be simplified by merely pointing to the particular assigned MSHR. If, however, all other data paths are still one word wide, the main operations would be essentially unchanged. Consequently, this extended interface would *not* significantly reduce the control complexity or the average lockout time delay per request.

The fields of the MSHR updated during memory receiver/input stack operations are the following:

1. In-input-stack indicators

2. Partial write codes

3. Number of words of block processed

4. Valid information indicator (being used indicator)

The in-input-stack indicators are set when the data word is written into the input stack and cleared when data is removed from the input stack and written into the cache buffer. The partial write code is set to indicate totally written when the data word from central memory indicates the cache buffer. In addition, whenever a data word is disposed of because of being totally written or having an obsolete MSHR, or is written into the cache buffer, the number-of-words-processed counter is incremented. On number-of-words-processed counter overflow (all words for a block have been received), the valid or used MSHR indicator is cleared.

For tag array control operations, the following fields of MSHRs are interrogated:

1. Input request addresses

2. Send-to-CPU indicators

3. In-input-stack indicators

4. Partial write codes

5. Valid indicator

6. Obsolete indicator

Fields 1, 5, and 6 are used along with current input request address and the n way MSHR comparator to determine if there is a hit on previously missed data still being handled (previous miss hit). Fields 2, 3, and 4 produce one of the following states for the previous miss hit:

- Partially written (Partial write code has at least one bit set.)

- Totally written (Partial write code is all 1's.)

- In-input-stack

- Already-asked-for (Send-to-CPU indicator is already set.)

Figure 3 indicates the actions followed by the tag array control under all the above combinations for a previous miss hit. On a miss, a MSHR is assigned, and the following is performed:

1. Valid indicator set

2. Obsolete indicator cleared

3. Cache buffer address saved in assigned MSHR

4. Input request address saved in assigned MSHR

5. Appropriate send-to-CPU indicator set and others cleared

6. Input identification tag saved in appropriate position

7. All partial write codes associated with assigned MSHR cleared

8. All MSHRs pointing to same cache buffer address purged (Set partial write code to all 1's)

Note that actions 5 and 6 will vary if the cache function was a prefetch (all send-to-CPU indicators are cleared, and no tag is saved). Action 8 prevents data from a previous allocation of a cache buffer block from overwriting the present allocation's data. On a miss and previous miss hit (the cache buffer block was reallocated for the same input address before all data was received), MSHR is set obsolete to prevent possible subsequent multiple hits in the MSHR comparator.

SIMULTANEITY

A previous miss hit on a data word just being received is definitely possible. Depending on the control operation, this word may have its corresponding send-to-CPU indicator's output forced to the send condition or may be read out of the input stack on the next minor cycle.

DIAGNOSABILITY

To diagnose this cache enhancement more readily, cache input functions should be added to clear and set the valid indicators of the MSHR registers. This would allow the following error conditions to be forced:

- Cache tag points to nonvalid MSHR register
- Multiple hit with MSHR comparator
- Previous miss hit status–totally written and not partially written

All other fields of the MSHR registers may be verified by using these special cache input functions in combination with the standard input functions with all combinations of addresses, identification tags and data.

CONCLUSIONS

This cache organization has been designed, prototyped, and verified. The design allows for the disabling of the MSHR registers. Using this capability, the direct effect of the number of MSHR registers on the execution times of a number of applications was noted. The reduced execution times of these applications directly demonstrated the effectiveness of this enhancement. (It is beyond the scope of this paper to analyze quantitatively the average lockout delay/request with respect to the number of enabled MSHR registers for different cache input rates and hit rates [cache buffer sizes]. This analysis will be reported in future work.) The cost of the 4 MSHR additions to the design was about 10% of the total cache cost.

ACKNOWLEDGMENT

The author thanks Control Data Canada for the opportunity to develop the new cache organization presented in this paper.

REFERENCES

[1] C. J. Conti. *Concepts of buffer storage*, *IEEE Computer Group News*, 2 (March 1969).

[2] R. M. Meade. How a cache memory enhances a computer's performance, *Electronics* (Jan. 1972).

[3] K. R. Kaplan and R. O. Winder. Cache-based computer systems, *IEEE Computer* (March 1973).

[4] J. Bell, D. Casasent, and C. G. Bell. An investigation of alternative cache organizations. *IEEE Transactions on Computers*, C-23 (April 1974).

[5] J. H. Kroeger and R. M. Meade (of Cogar Corporation, Woppingers Fall, NY). Cache buffer memory specification.

[6] A. V. Pohm, O. P. Agrawal, and R. N. Monroe. The cost and performance tradeoffs of buffered memories. *Proceedings of the IEEE*, 63 (Aug. 1973).

[7] A. J. Smith. Sequential program prefetching in memory hierachies, *IEEE Computer* (Dec 1978).

[8] G. H. Toole. Instruction lookahead and execution traffic considerations for the ______ cache design (Development division internal paper), Control Data-Canada, 1975.

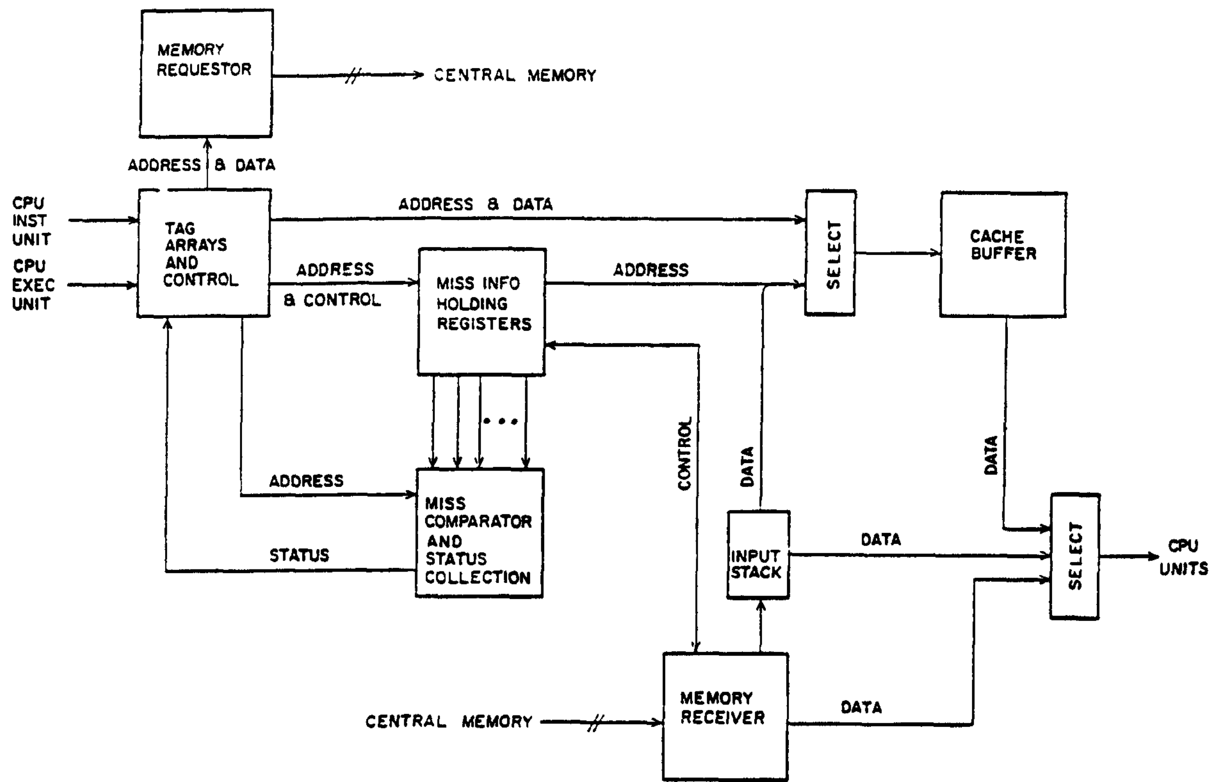

Figure 1. Cache Organization

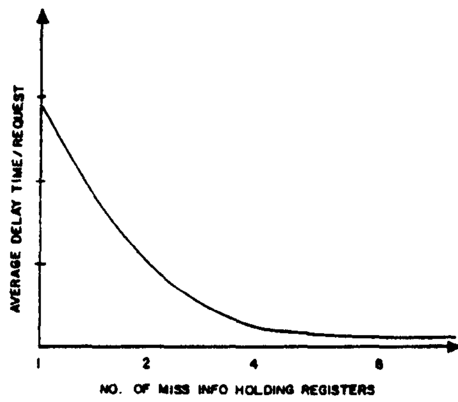

Figure 2. Qualitative Curve for Lockout Delay

INPUT FUNCTION	PARTIALLY WRITTEN	TOTALLY WRITTEN	IN INPUT STACK	ALREADY ASKED FOR	ACTION
READ	NO	NO	NO	NO	SET SEND-TO-CPU BIT SAVE IDENT
READ	NO	NO	NO	YES	READ FROM CENTRAL MEMORY (BY-PASS)
READ	NO	NO	YES	X	READ FROM STACK
READ	YES	NO	X	X	READ FROM CENTRAL MEMORY (BY-PASS)
READ	YES	YES	X	X	READ FROM CACHE BUFFER
PREFETCH	X	X	X	X	NO ACTION
WRITE	X	X	X	X	WRITE BYTES TO CACHE BUFFER. SET APPROPRIATE PARTIAL WRITE BITS.

WHERE X IS DON'T CARE

Figure 3. Previous Miss Hit Operations