COMPANY CONFIDENTIAL

TITLE: PDP-11 CACHE MEMORY SCHEME & CONSIDERATIONS

PDP-11/40 Tech Memo #20
Date: October 6, 1970
Author(s): Don Vonada, Jim Murphy
Revision: None **Obsolete:** None
Index Keys: Cache Memory, Buffer, Memory, Memory Processor Interaction, Paging/Segmentation

Distribution: PDP-11 Master List
PDP-11 Coordinating Committee
PDP-11/40 Group

ABSTRACT

The parameters of a buffer memory to be used in a PDP-11/40 are discussed. The objective is to reasonably constrain the variables involved so that a simulation program may be written and the proposed buffer performance can be evaluated.

The parameters of major interest are cache size, block size, number of sets, and write-back algorithms. The present guess is that the cache will likely be about 512 words in 8 word blocks, with 2 or 4 sets of congruently mapped blocks.

1. PHYSICAL PARAMETERS

1.1 Buffer Size - Buffer size refers to the total word capacity of the cache memory, irrespective of the mapping algorithm. (Refer to Figure 1.) Buffer sizes of 256, 512, and 2048 16-bit words will be simulated. These values were chosen to provide insight and direction. The final choice is not limited to these values. It is interesting to note that the IBM 360/85 employs a cache with 16K 8-bit bytes, expandable to 32K bytes.

1.2 Block Size - Block size refers to the number of words transferred from main to cache memory for each word requested which is not in cache memory. Transferring a number of words for each request is normally done to give a pseudo prefetch capability to the buffer. Block sizes of 4, 8, and 32 16-bit words will be simulated. The general comments of Section 1.1 are applicable. Again, using IBM as a guide, their block size is 64 8-bit bytes. The block size of 1 will be tried for one configuration to provide a basis for comparison with the simulation work done for the PDP-8F. This is expected to be a relatively unoptimum solution because of the absence of prefetch and the requirement of an associative address bit with each cache word.

1.3 Mapping Algorithms - (Refer to Figure 1)

1.3.1 Full Associative - This method makes the most efficient use of the buffer space. However, address tags are larger than other schemes thus increasing hardware costs. Also, all buffer addresses must be checked, thus decreasing buffer performance.

1.3.2 Congruent - Congruent mapping restricts the transfer of blocks of a core memory area partitioned by rows and columns to one block of buffer memory which corresponds with the one (and only one) row of core memory. A fewer number of buffer address (tag) bits

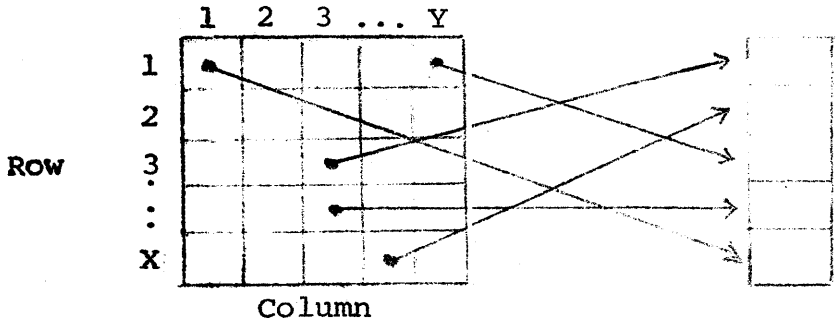
are necessary since the desired block is either in a specific row of the buffer or it must be requested from core memory. Hardware costs are reduced and performance is improved. Congruent mapping will be simulated as a special case of Set Associative Mapping (Section 1.3.3) where there is only one "set."

1.3.3 Set Associative - Set Associative mapping is similar to Congruent with the added sophistication of allocating two or more buffer blocks to the same core memory row. This technique reduces the conflict of contention for the same block in buffer memory when the program references two or more blocks in the same row of core memory. Simulation will be performed for sets of 1, 2, and 4 blocks of buffer memory per row of core memory.

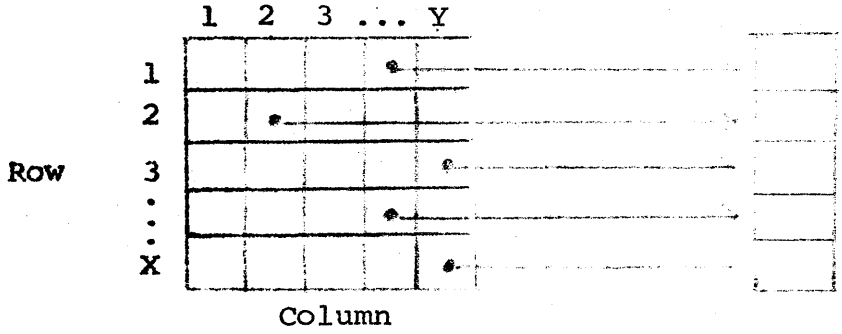
1.3.4 Hybrid - Any combination of the above mapping algorithms is a hybrid. IBM 360/85 uses the Full Associative algorithm to map large sections of core (1K bytes) onto the buffer memory, but not all bytes are loaded into the buffer. Blocks of 64 8-bit bytes within the mapped section (called sector) are Congruently mapped and a control bit is set signifying which block of the sector contains valid information. Consideration of a hybrid system is pending the results of the simulation of the more straight forward methods.

1.3.5 Special - The idea of mapping instructions separate from data was considered and rejected on the ground that the PDP-11 instruction architecture does not complement this scheme. Most of the PDP-11 instructions are multiple word instructions, thus it is typical for data to immediately follow the instruction. Separating the instruction and data into separate maps does not appear to have any advantage, especially if more than one word per block is used as in 1.2.

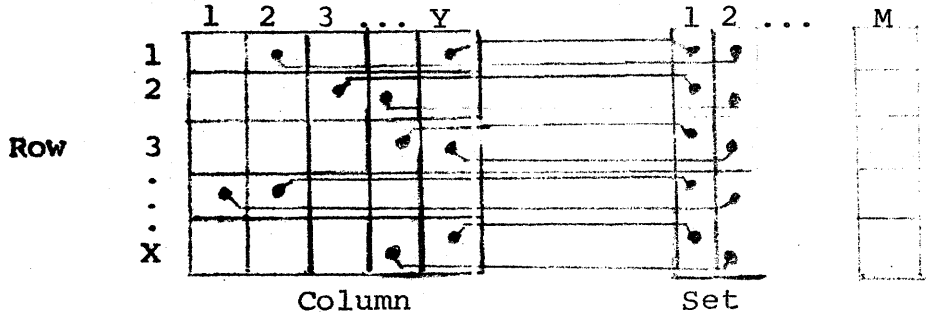
MAIN MEMORY CACHE MEMORY



FULL ASSOCIATIVE
Any core memory block maps into any cache memory block.



CONGRUENT
Any block from one row of core memory maps into a row dedicated cache memory block.



SET ASSOCIATIVE
Given M as the number of sets, up to and including M blocks from one row of core memory may be mapped into any row dedicated block of cache memory.

NOTE: Each square represents a block of N words in both main and cache memory.

Buffer capacity is the product N · M · X.

FIGURE 1-MEMORY MAPPING SCHEMES

1.4 Program/Buffer Ratio - Program/Buffer Ratio is defined in this paper as the ratio of the number of core blocks containing program information to the total number of buffer blocks. There was some concern that the simulation parameters may be biased such that the test program would always fall into one column of the core memory matrix. It is necessary to insure that this does not happen. The program should reside in a number of columns to increase the likelihood of contention for one buffer block.

If there is no discontinuity in the program address space, the number of core blocks is easily defined as:

$$N_c = \frac{M}{B}$$ where M = Program size (words)
B = Block size (words)

Similarly, the number of buffer blocks is defined as:

$$N_b = \frac{A}{B}$$ where A = Buffer size (words)

Program/Buffer ratio is therefore:

$$Nc/Nb = M/A$$

or simply the ratio of program size to buffer size. This ratio should be calculated and used as an indicator for all programs used in the simulation.

2. PLACEMENT & PURGING ALGORITHMS

2.1 Core Image Update - It is intuitive, considering the influence of I/O transfers and multiprocessors, that cache memory and the blocks of core that it represents be exact images of one another at all times. This is philosophically ideal, but hardware-wise extremely difficult and maybe even impossible if the overall performance of the system is not to be significantly degraded.

Except for consideration in Sections 2.1.3 and 2.1.5, the core image update schemes outlined below will have no bearing on the operation of our simulated cache system.

2.1.1 Write back an entire block at replacement time. Core is not updated until its associated block is replaced in cache memory. At that time the entire block is written to core memory.

The problems inherent to this scheme are:

1. Data channel outputs concerned with data updated in cache memory must avoid passing on stale data.

2. Multi-processors must somehow avoid accessing stale information that is used to control synchronization between themselves.

3. The processor must delay the amount of time necessary to transfer an entire block to core before the new block can begin to be transferred to the cache.

2.1.2 Write back all modified words of a block at replacement time. This scheme also has problems 1 and 2 of the scheme in 2.1.1. The delay at replacement time to write to memory is decreased as non-modified registers are not transferred.

2.1.3 Immediate core update on stores to cache. This method has the obvious benefit of keeping core memory as current as possible thus avoiding problems with:

1. Data Channel outputs
2. Multi-processor synchronization
3. Block updates at replacement time

However, it has the negative effect of holding up cache memory cycling on each store that is preceded within one memory cycle by another store. The delay is until the memory cycle has completed. IBM, on the 360/85, has given the processor the capability of buffering one instruction requesting the storing of information into core memory in an attempt to improve this situation.

The FORTRAN program, to simulate cache memory systems, will also keep track of the number of cache memory cycles that would be gained if a one word buffer were utilized as intermediate storage to core memory. Any improvement caused by the one word buffer will be a function of:

1. The proximity of stores.
2. The replacement of any block in cache memory which necessitates the emptying of the one word buffer before the replacement can take place.

The simulation will assume that 1Ø cache cycles equal 1 memory cycle and if a second store occurs within 1Ø cache cycles of the preceding store, the cycle gain will be registered as 1Ø minus the number of intervening cycles:

e.g. let S_n = store cache cycle
x = non-store cache cycle

S_1XXXS_2 the gain here would be 6 cache cycles as S_2 could be buffered rather than delayed until the S_1 memory cycle was completed.

The simulator must also factor in the effect of strings of stores:

e.g. $S_1XXXS_2XXS_3XXXS_4$

Without a buffer, the delays would be:

D_2 = 6 cache cycles
D_3 = 7 cache cycles
D_4 = 6 cache cycles

However, the use of a buffer will not gain 19 cache cycles because there would still be delays at S_3 and S_4 due to the buffer being occupied:

D_3 = 3
D_4 = 6

Therefore, the actual gain would be 1Ø cache cycles.

As block replacements require the buffer to be emptied before proceeding, this must also be considered.

e.g. let R = start of block replacement

$S_1XXXS_2XXS_3XXXS_4XXR_5$

Without a buffer, the delays would be

$D_2 = 6$
$D_3 = 7$
$D_4 = 6$
$D_5 = 7$

With a buffer, the delays would be

$D_3 = 3$
$D_4 = 6$
$D_5 = 17$ (to complete the writing of S_3 which began at S_4 and then write S_4)

showing no actual gain by using a one-word buffer. Note that the cycle flows chosen were purely for explanatory reasons and that it will be the task of the analysis program working with actual flows to determine the cost performance benefits of employing a one word buffer.

Another pertinent point is that the use of an intermediate buffer causes the immediate core update scheme to no longer be immediate in the sense that core is a true image of the cache at all times. This brings the problems of:

1. Data channel outputs, and
2. Multi-processor synchronization

to the limelight again.

A thought to put away for future consideration is that the increase of buffer words from 1 to N may really pay off in applications where a significant amount of floating point calculations are performed. As the processor time for a floating point operation is relatively long, this time would be available for the emptying out of buffers that had been previously filled.

2.1.4 Core modifications to cache. The inverse of the update of core with current cache data is the updating of the cache with current core data that came to be via:

1. Data channel inputs, and
2. Multi-processor stores.

Again, this does not affect our simulation, but certainly must be considered and solved in the hardware.

2.1.5 No cache activity if store not in cache. Whenever a store is requested, to a block not currently in the cache, the store will be made directly to core memory requiring no cache activity whatsoever. This is a significant point relative to the simulated cache system as it becomes a major qualifying factor in determining when block replacements are to be made and thus, in the performance of the total system under evaluation.

2.2 Purging Algorithms. The algorithm used to select the buffer block into which the next core image is to be placed is usually referred to as a purging algorithm. A number of them are listed below and unless there is strong feeling to the contrary, 2.2.1 will be used for the simulation. It is relatively easy to implement and again, the 360/85 uses it. The rest are listed for information.

2.2.1 Least recently used.
2.2.2 Least frequently used.
2.2.3 FIFO; First in, First out.

2.2.4 Random

2.2.4.1 Pure

2.2.4.2 Weighted; block presently being used and/or perhaps most frequently used is not purged.

3. CACHE IN A PAGING SEGMENTATION ENVIRONMENT.

It is becoming increasingly obvious that the interaction of the Paging and Segmentation hardware is intimate with the buffer memory. In focusing attention on the problem, two configurations were considered: 1) Buffer next to core memory, and 2) Buffer next to CPU.

3.1 Buffer between core memory and the P/S hardware. The disadvantage of this scheme is that the buffer tag address must contain the extra bits necessary to specify all possible core addresses and thus require extra hardware to implement. The advantage is that in a time sharing environment the monitor can reside in a fixed physical address so when users are swapped in and out of core, the monitor image in the buffer is not changed.

3.2 Buffer between the CPU and the P/S hardware. This scheme has the advantage of being fast and requiring only enough tag address bits to specify the virtual memory. However, the disadvantages are that every change to the P/S hardware would require an update of buffer memory. Also, core memory modifications from sources other than the CPU (such as I/O and multi-processor) would be very difficult to reflect in the buffer memory, the P/S hardware would have to check the memory address as well as flag the buffer memory, if applicable.