To: Distribution

From: Bernard S. Greenberg

Subject: Proposal for Multics Cache Software

Date: March 21, 1974

This MTB describes a proposal for the implementation of software for use of the 6100 Cache Store System with Multics. The Cache and the advantages to be gained by its use are described. The problems associated with its use on Multics are described, and solutions proposed.

WHAT IS THE CACHE, AND WHY SHOULD WE USE IT?

The 6100 Cache Store System is a fast, random-access buffer memory, added to a 6000-series processor, at its port logic. It is significantly faster than main memory (50 ns. basic access time, as opposed to 1.2 us), and is a physical part of the processor, as opposed to the memory hierarchy. It attempts to keep copies of recently used data from in itself, in much the same way that Multics attempts to keep copies of recently used pages in main memory. Thus, if the processor's various subunits request some data from memory which has a copy in the cache, the request will be satisfied by the cache unit, and no request to main memory will be made. This saves a great deal of time.

The cache thus forms the fastest level of a multilevel memory hierarchy. Core, paging device (bulk store), and disk would be the next levels in a Multics system with the cache. The successful use of a cace depends on the patterns in which programs reference data, and those features (spatial locality of reference) which facilitate this use are common to most programs. Simulation studies by Honeywell predict that the use of the 6100 Cache Store System on a model 6070 (No EIS, no segmentation, paging, or ring hardware), with four-way interlace and half-microsecond core, would reduce the CPU time required by a GCOS COBOL compilation by 40%, as opposed to the same system without the cache. As there is no reason to suspect that typical

Multics programs are that radically different from GCOS programs, in their memory reference patterns, some comparable performance improvement would not be unreasonable.

HOW IS THE CACHE ORGANIZED AND HOW DOES IT WORK?

The 6100 Cache Store System consists of 2048 36-bit words on RAM bipolar chips. These words are organized into 512 blocks of four words each. Each block can contain four consecutive words, from a mod-4 boundary, from main storage. These blocks are further organized into 128 'columns', each containing four blocks. The blocks in any single column can hold only certain blocks from main storage. Thus, column 0 can hold block 0, block 128 block 256, block 384, etc. of main store. In general, column n can hold any block n+128*m, for any m. This in effect divides main memory into 128 equivalence classes of four-word blocks competing for cache blocks in the same column. The cache works by absolute address - it determines which address is being referenced in main memory from the 24-bit final address presented to the port logic by the Appending Unit. Thus, association of contents with addresses is oblivious to segmentation. Associated with the blocks of cache memory is a directory of 512 entries, each entry describing the contents of a corresponding cache memory block. A single bit (FULL/EMPTY) describes if the corresponding block contains valid information. A fifteen-bit address field contains the upper 15 bits of the twenty-four bit absolute address of the words in main memory to which the words in the corresponding cache block correspond. The next seven bits of this address are implicit, as they define the column in which this block appears. The remaining two bits (24=15+7+2) select a particular word of the four word block.

The cache is interposed between the Appending Unit, which in a Multics processor mediates memory requests, and the port logic of the processor, which is responsible for directing a memory request (fetch, store, set a mask, etc.) and an absolute address out of some processor port, based on the upper bits of the absolute address, and either receiving data from or sending data to the selected controller. Among the functions of this logic is the sending of a request for service to the selected controller, and receiving a reply when the request has been granted. The processor module which initiates the request suspends its operations until this acknowledgement is received.

The cache unit intercepts all of these memory requests. If the request is a simple read (<u>NOT</u> a lock-type 'read and clear', as would be used by the stac, stacq, ldac, ansa, etc., instructions) the cache inspects the directory entries for that column of cache which would contain the words being requested. This means that four comparisons of the four 15-bit address fields are done in parallel. The cache unit does not let the request for service go to the memory controller until it verifies that none of the four addresses compare. (A FULL bit being off prevents a successful comparison as well.) If a match is found, the cache retrieves the stored words from the cache block whose directory entry matches the requested address, and notifies the Appending Unit that the data is available. Not only does this retrieve the data with great rapidity, but enhances multiplexing of the memory controller. If no match is found, the request proceeds to the memory controller as usual. However, the cache replaces the contents of some block in the appropriate column with the two words which are returned from the memory controller. What is more, the cache unit makes a <u>second</u> request on the port logic to fetch the other two words to be stored in the cache block. The acknowledgement is then given to the Appending Unit. The block to be replaced is selected on a first-in-first-out (FIFO) basis. This is implemented by a two-bit counter associated with each column of the cache directory.

Write-type requests go directly to main storage. Hence, the cache never fetches words when a store-type instruction is issued. However, a compare is made with each of the directory entry address fields in the appropriate column, and if a match is found, the cache unit updates its contents with the data being written.

Locking-type fetches (read-and-clear) also go directly to main memory. The cache also purges the contents of a matching block, if any, by turning off the FULL/EMPTY bit.

The cache can be turned on and off by a bit in the CPU mode register, which is loaded by the LCPR instruction. Certain faults also cause the cache to turn off. A special register exists for controlling the modes of operation of the cache, controllable by another variant of the LCPR instruction. Every time the cache is turned on, it clears all of its FULL/EMPTY bits to EMPTY, as it has no way of knowing what was stored in main memory while it was off, and it should never contend that any data that it might return are valid.

Two features have been added to the cache specifically for Multics. An SDW bit has been defined, which, when off, tells the Appending Unit to inhibit the cache from making any successful address comparison. This has the effect of allowing some segments to have words sorted in the cache and others not. Also, a special variant of the CAMP (clear associative memory of PTWs) instruction has been defined which clears the FULL/EMPTY bit of any block in any column whose address in its directory entry matches an address given as part of the instruction. This allows clearing of an entire page out of the cache in two instructions. As each one of these instructions must cycle through 128 columns, doing the four-way compare with each, 25 microseconds is required for this operation.

A special version of the CAMS (Clear associative memory of SDWs) instruction has also been defined which turns off all of the FULL/EMPTY bits at once, invalidating the entire contents of the cache at one.

WHAT ARE THE PROBLEMS WITH THE USE OF THE CACHE WITH MULTICS?

The cache is an integral part of the processor - it is a buffer between the processor's component units and main memory. In Multics, many active modules (processors, IOM, Bulk store Unit, etc.) are connected to main memory. Any of these active modules may inspect or change the contents of amin memory. Hence, the cache unit in a processor may hold copies of words of main memory which have been changed there by the IOM or another processor since they were loaded into the cache.

Thus, system activity which involves more than one active module having access to any given word of core must be reconsidered in light of the cache. Since write-type requests by a processor go through the cache, directly to main storage, a word of cache storage never contains a word whose contents are more recent than the corresponding word of main storage. Difficulty arises only when a processor cache holds a word less recent than a copy in main storage. This can only happen for main memory words which are either 1) capable of being written into by a non-processor active module, or 2) accessible to two processors, at least one of which can write them. The use of segmentation allows the grouping of data with various access attributes into segments (e.g., only those segments known as 'I/O buffers' contain words which are subject to being written by another active device than a processor leaving the issue of paging I/O aside for a moment). The identity of segments which can be written by one processor and seen by another (assuming for the moment that the processors are to stay in the same processes) is also an issue which can be handled easily. Hence, segmentation provides a simple means of identifying words which should be treated in different ways with respect to the cache. Thus, the bit "sdw.cache" has been defined, whose absence in an SDW forces all requests for data from the corresponding segment to go directly to main memory, and prevents the cache from loading data from this segment into itself. Such "non-encacheable segments" are transparent to the cache, that is, the cache may never contain words from them and hence may never have a copy more recent than the copy in main storage.

The remaining issues are the determination of which segments are encacheable, and when the cache must be cleared.

THE ENCACHEABILITY OF SEGMENTS

From the considerations stated above, it should be clear that I/O buffer segments (e.g., tty_buf, tape_data) are not encacheable. Thus, any SDW made for these segments should prevent words from these segments from entering any processor's cache, as teletypes and tape controllers (via the Datanet and the IOM) may update words in main storage which the cache could not observe.

Similarly, it is clear that data bases which are accessible to only one processor, i.e., the prds's of processors, are always encacheable. Even though prds's are created by some other processor than the processor for which they are intended (for all cases other than the bootload processor), there is never any difficulty in the intended processor not seeing a change made by the other processor, as the former has not started to run by the time the latter is finished with its prds.

Pure procedure segments are always encacheable. Again putting off the issue of how these segments were written, initially, no module may write in these segments, and hence, a cache copy can never be less recent than the legitimate contents of the segment.

In order to handle all other cases, the concept of 'per-processor' can be mapped into that of 'per-process'. As long as the process-processor pairing remains unchanged, the segments which are non-encacheable (other than those already mentioned) are those which are writeable by one process and readable by another. Hence, stacks, linkage sections, pds's, kst's, and programs are all encacheable. Initializer and answering service data bases are encacheable. Inter-user data bases in outer rings, and data bases such as the SST, tc_data, and descriptor segments themselves are not encacheable. Any data base which is written only at system initialization (by the bootload processor) and read thereafter can never be changed after a non-bootload processor can fetch its words into its cache; hence these segments (e.g., sys_info) are encacheable.

MAKING THE ABOVE STRATEGY WORK, AND THE CLEARING OF THE CACHE

Although the above strategy provides a scenario whereby most heavily-used segments are encacheable, it is founded upon at least one unreasonable assumption, and two unstated considerations.

The process-processor pairing can not &e assumed to remain constant. A processor may pass through many processes.

Similarly, a process may be viewed as possibly passing through several processors. It is this latter view which provides the key for making the process-processor pairing work for the cache. If a process never changes processors, the words of its per-process encacheable segments (writeable data bases) will never appear in the caches of other processors, as we have stated that these segments are not readable by any other process. Similarly, no other processor may modify the words of this segment (assuming no read permission implies no write premission). Hence, the copies of words of this segment in the cache of the processor which has been running this process can never be less than up to date. If this process now switches processors, however, the new processor will acquire copies of these words in its cache. Still, there is no problem. Should the process switch back to the original processor, however, copies of words of the segment of interest are still in the original processor's cache, and are less recent than possible modifications made to the copies in main memory by the other processor. The solution is simple: Every time a process switches processors (easily determined from data kept in the APT), the new processor should invalidate the contents of its cache.

The first unstated consideration is the determination of which segments are writeable to one process and readable to another. Scanning ACLs is one solution. A more reasonable solution, which depends upon considerations stated in the ext section, is to observe the creation of SDWs for a particular segment. As long as only read or read-execute-allowing SDW's are created for a segment, the segment is encacheable, and all of these SDWs reflect this. Similarly, is there is but one SDW for the segment (i.e., only one process is currently using it) it is encacheable, even if this SDW permits writing. If, at any time, a segment in either of these states changes into the non-cacheabilitytate where one write-access and one read-access SDW exist, ALL SDW's must be changed to reflect the non-cacheability of the segment. Furthermore, the caches of all of the processors must be cleared, to remove copies of words of this segment. This can be done via a modification of the current connect-fault mechanism, setting special flags so that a processor receiving a connect fault will interpret the latter as a signal to clear its cache.

SDW's are modified or destroyed by three mechanisms other than the creation mechanism of the segment fault, considered above. Deactivation (resulting from reuse of AST entries or segment destruction) destroys all SDW's for a given segment. As part of deactivation, all of the pages of a segment are forced out of core. As explained in the next section, this will clear the segment out of the system's caches page by page. Access changes cause an operation known as a 'setfaults' to be performed on all existent SDW's of a segment. This causes later segment faults which will recalculate each process' access to the segment. The encacheability of the segment will be recomputed at those times. However, at the time the setfaults is done, the segment's words must be driven out of the system's caches, if it is currently encacheable. This may be done via the same connect fault mecahnism as before. Termination of segments causes SDW's to be destroyed. While this can never make an encacheable segment non-encacheable, it may do the reverse, granting encacheability at the time the last write-access SDW of a segment is destroyed. This could prove to be an important consideration in the case of a library procedure being updated by library maintenance personnel. If cacheability were not granted at the time the library maintenance personnel terminated the segment, ordinary users would not use the cache for this segment.

Information concerning the nature of SDW's of a given segment is best kept in the AST entry for that segment, and in a per-SDW data base known as the 'system trailer', each of which have enough free bits to record the necessary information for dynamic encacheability computation.

The encacheability of hardcore segments is best determined from an SLT attribute, which comes from a system header attribute, defining the encacheability of hardcore segments at system generation time.

The modification of SDW's by bounds faults is not of interest, as it changes neither the mapping of segments into nor access to segments.

The second unstated consideration is paging.

PAGING

The remaining important issue is reflecting changes in the contents of main memory due to paging to the system's caches. The proposed solution relies on the fact that no program other than page control references a page frame of main memory between the time that access to the page frame is turned off and the time that access to the page frame is opened with the contents of some other page of the virtual memory in this frame. This turning off of access occurs at the time that page control decides to replace a page, due to either lack of recency of use, or deactivation. Between the closing of access and the new opening of access, data may be read into this page frame from the paging device or disks, and page control may inspect or modify the contents of this page frame. This modification and inspection are done with special non-encacheable segments know as 'abs-segs', which are essentially segments defined to be whatever it is desired to look at or modify at the time that it is desired to do such. Hence, page control will always see the correct contents of the page. At the time that access is closed to a page, all of the processors in the system are notified to use the special verison of the 'CAMP' instruction described above, to clear all words of this page out of their caches. Hence, by the time access is opened to the new page in that page frame, there will be no words of the old contents in any processor's cache.

OTHER CONSIDERATIONS

Directories are not encacheable. This is because they are inspected and modified at segment deactivation time via the use of a dynamically created SDW (resulting from the use of an abs-seg). Were the directory being referenced in this way encacheable, copies of words in any cache would not see changes made in this way. Alternatively, one could clear system caches and redefine the directory's encacheability at this time, but this seems a fairly expensive approach, especially in view of the likelihood that directory referencing patterns do not warrant the encacheability of directories.

The cache is not turned on until processor initialization, which, for the bootload processor, is late in system initialization. This avoids having to deal with several memory management policies used only during initialization.

Ring-zero patching must clear all system caches, for it violates the encacheability rules for any read-only segment or single-process segment that it patches.

Abs-segs in general are non-encacheable. These 'segments' are used to extend a process' address space to encompas main memory pages or segments (i.e., other processes' descriptor segments) not accessible by other means. Were abs-segs encacheable, they might bring words into the cache of segments or pages which have already been removed from the cache of this processor, or established as not being in the cache of this processor. Furthermore, abs-segs are frequently used to inspect the result of I/O operations, which the cache never sees.

The cache must be turned off by the BOS toehold, when BOS is entered, or preferably before (to allow BOS to dump the cache as it was at the time some system problem occurred). The swapping of main memory images done by BOS necessitates that the cache be either off when BOS is entered, or maintained by BOS. BOS should not use the cache, as the performance of BOS was never an issue. Also, the cache should be turned back on (implicitly clearing it) as processors are restarted from a system-trouble interrupt (the return-to-BOS-or-halt mechanism).

There must be BOS software to dump the cache, both to the printer/tape and to the DUMP partition. Similarly, the FDUMP printing mechanism must be rewritten to dump this information.

Multitasking plays havoc with many of the strategies described above, starting right at the process-processor pairing assumption. If multitasking is ever implemented, some user-controllable segment attribute describing inter-task sharing must be defined (probably for other reasons as well), which would then be used in determining encacheability.

INVITATION

Address all comments to:

Bernard Greenberg
CISL
575 Tech Square