A New Solution to Coherence Problems in Multicache Systems

LUCIEN M. CENSIER AND PAUL FEAUTRIER

Abstract—**A memory hierarchy has coherence problems as soon as one of its levels is split in several independent units which are not equally accessible from faster levels or processors. The classical solution to these problems, as found for instance in multiprocessor, multicache systems, is to restore a degree of interdependence between such units through a set of high speed interconnecting buses. This solution is not entirely satisfactory, as it tends to reduce the throughput of the memory hierarchy and to increase its cost.**

A new solution is presented and discussed here: the presence flag solution. It has both a lower cost and a lower overhead than the classical solution. A very important feature of this solution is that it is possible, in a cache-main memory subsystem, to delay updating the main memory until a block is needed in the cache (nonstore-through mode of operation).

Index Terms—**Caches, coherence, memory hierarchy, multiprocessor systems, nonstore-through.**

I. Introduction

THE IDEA that a computer should use a memory hierarchy dates back to the early days of the field. There is for instance, a suggestion to this effect in the classical paper of von Neumann *et al.* [11]. A hierarchy is useful because the access time of main memory increases with its size. As soon as a certain capacity is required, the memory is inherently slower than the processor and becomes the bottleneck in the system. By adding a small memory which fits the processor speed, one may expect a considerable increase in performance if this memory is cleverly used.

The first system in which this process was automated was the ATLAS demand paging supervisor (Fotheringham [5]). The ATLAS hierarchy had two levels: a core memory and a drum. As the drum latency time is of the order of several milliseconds, it was possible to implement the supervisor as software modules.

The first proposal to apply similar techniques to fast levels was by Bloom *et al.* [2]. After a variety of theoretical studies of which Wilkes [12] and Opler [9] are examples, the first implementation of the idea was the IBM 360/85 (Gibson [6], Conti *et al.* [4], Liptay [7], Conti [4]). The resulting device, the cache memory, is now a component of most computers in the medium to high performance range.

In such a system, all data are referenced by their main memory address. At any given time, a certain subset of the contents of the main memory is copied in the cache. One says that such a datum is present in the cache. If a processor reads a datum in this subset, then the corresponding value is returned without referencing the main memory, after a delay which is of the order of one processor cycle time. This event is called a "hit." A directory records the addresses of all data which are present in the cache. To reduce the size of this directory, the main memory and the cache are divided in equal sized "blocks," all bits of a block being simultaneously all present or all absent from the cache. The block is then the allocation unit in the cache and also the minimum amount of data which may be transmitted between the cache and main memory. A combination of hash-coding and associative technique is used to implement a very fast search algorithm in the directory (see for instance the discussion in Bell *et al.* [1]).

In addition to the main memory address, the directory may include several flags per cache block. The VALID flag, when set, indicates that the corresponding block does hold the latest information associated to its main memory address. It is reset when the contents of the cache are undefined (for instance, at Initial Program Loading Time). Some designs include a MODIFIED flag. When set, it indicates that the block has been modified by the attached instruction processor.

The effective access time of a cache system depends critically on the hit ratio, i.e., on the probability that a requested datum is present in the cache. This in turn depends on the proper selection, by a replacement algorithm, of the cache content. A block is copied in the cache only when found absent after an access from the processor (a "miss"). This means that another block must be expelled. The usual choice is the Least Recently Used block (among a group of blocks with the same hash-code). Obviously, when the cache contains blocks with their VALID bit reset, these are used up before any valid block is expelled.

Two quite different modes of operation have been proposed for the processing of STORE accesses. In the store-through mode, a modified datum is always written in main memory and is written in the cache only if it is already present there. This mode is used in most systems (IBM 370/168, etc. ···). In the nonstore-through mode, LOAD and STORE accesses are treated alike: if the block to be written into is absent from the cache, then it is copied from main memory. All subsequent accesses to this block, whether read or write, are processed by the cache, until such time as it is selected by the replacement algorithm. At this time, it is

Manuscript received April 27, 1976; revised April 11, 1978. This work was supported in part by the Institut de Recherche en Informatique et Automatique under Contract 74/185.

L. M. Censier is with CII-Honeywell Bull, Les Clayes-sous-Bois, France.

P. Feautrier is with Université Pierre et Marie Curie, Paris, France.

0018-9340/78/1200-1112$00.75

written back to main memory. This step may be bypassed for blocks which were not written into while in the cache, if a MODIFIED bit is implemented. The net result is that a single memory access by a processor may induce zero, one or two accesses to main memory, thereby complicating the timing of the data access algorithm.

The advantage of the nonstore-through mode is that, at least in theory, the access rate to the main store may be reduced to any desired value by a sufficient increase of the size of the cache. In contrast to this, in the store-through mode, the access rate to main memory cannot be lower than the write access rate of the processor. Examination of instruction mixes shows that, depending on the processor architecture, from one tenth to one third of all accesses are STORE accesses. This figure is then a lower limit for the miss ratio of a cache in the store through mode. A simulation analysis which supports these views is reported in Bell *et al.* [1].

Both modes of operation run into coherence problems when applied to multiprocessor systems. A memory scheme is coherent if the value returned on a LOAD instruction is always the value given by the latest STORE instruction with the same address. There is obviously no coherence problems in a memory hierarchy with only one access path between each level. This however causes technical problems in high performance systems. The unique access mechanism would have to be prohibitively fast. Furthermore, a cache must be closely integrated to its processor to avoid transmission delays. I/O processors, on the other hand, have data rates substantially lower than instruction processors. There is no element of locality in the data addresses they issue and therefore no performance advantage in connecting them to a cache. This induces new coherence problems.

To take a specific example, let us consider the simple case of a biprocessor system with two caches in the nonstore-through mode, the main memory being shared by both processors. Let T_1 and T_2 be two tasks running on processors P_1 and P_2 with caches K_1 and K_2. Let a be the main memory address of a block which is read and modified by both tasks. One may assume that T_1 and T_2 are correctly programmed: for instance, that all modifications of the contents of a are protected in critical sections. A modification of the contents of a by T_1 is done in K_1 but is not transmitted to main memory; in consequence, a subsequent LOAD by T_2 will find an obsolete value of a.

Another difficulty arises when one task T may be executed by P_1 or P_2 depending, for instance, on the time of arrival of external interrupt signals. It may happen that a has a copy in both caches; in this situation a modification to a executed in K_1 is not reflected to K_2. After a processor switch to P_2, T will obtain an obsolete value of a. This example shows that there may be coherence problems even if no datum is shared between tasks.

It is clear that the store-through mode is not sufficient in itself to insure coherence. In the example above, on a LOAD instruction, neither processor will access main memory and modifications to the contents of a by P_1 and P_2 will be entirely uncoupled.

Evidently, a solution to the coherence problem implies the invalidation of blocks when there is a risk that their contents have been modified elsewhere in the system. One may use total preventive invalidation on carefully selected events: task switches, exit from critical sections, etc. ···. One may prove that this suffices to insure coherence in the absence of programming errors. This solution, however, will greatly decrease the hit ratio of the cache, and is not suited to high performance systems.

In another solution, addresses of modified blocks are broadcast throughout the system for invalidation. This is the classical solution and will be studied in the next paragraph. We will then describe a new solution in which the frequency of invalidation order is reduced by keeping tabs on the whereabout of block copies.

II. The Classical Solution

This solution is found in biprocessor systems or in monoprocessors with an independent I/O processor. These systems use the store-through mode.

To insure coherence, every cache is connected to an auxiliary data path over which all other active units send the addresses of blocks to be modified. Each cache permanently monitors this path and executes the search algorithm on all addresses thus received. In case of a hit, the VALID bit of the affected block is turned off.

The drawbacks of this solution are the following.

1) The invalidation data path must accommodate a very high traffic. The mean write rate for most processor architectures is between 10 and 30 percent. For some instructions, the peak rate is much higher: 50 percent for a long move and 100 percent for a move immediate. If the number of processors is higher than two, the productive traffic between a cache and its associated processor may be lower than the parasitic traffic between the cache and all other processors. This explains why the classical solution has been confined to systems with at most two caches.

2) Unless special precautions are taken, the cache will spend most of its time monitoring the parasitic traffic. The usual way out of this problem is to duplicate the cache directory. There is no need to interlock accesses to the two copies unless a modification of the directory is required: this is a comparatively rare event.

3) To accommodate the peak invalidation traffic, one may have to insert a small buffer to queue up addresses of modified blocks. There is a small probability of noncoherence if a read request by processor P_1 is executed between a modification by the other processor and the actual invalidation in P_1 cache. This phenomenon may or may not occur depending on such parameters as the relative timing of the cache and main store, priority schemes, etc. The resulting very low frequency inconsistencies are probably ascribed to nonreproducible hardware errors.

III. The Presence Flag Technique

The objective of this method is to reduce the coherence overhead by filtering out all or almost all uneffective invalidation requests.

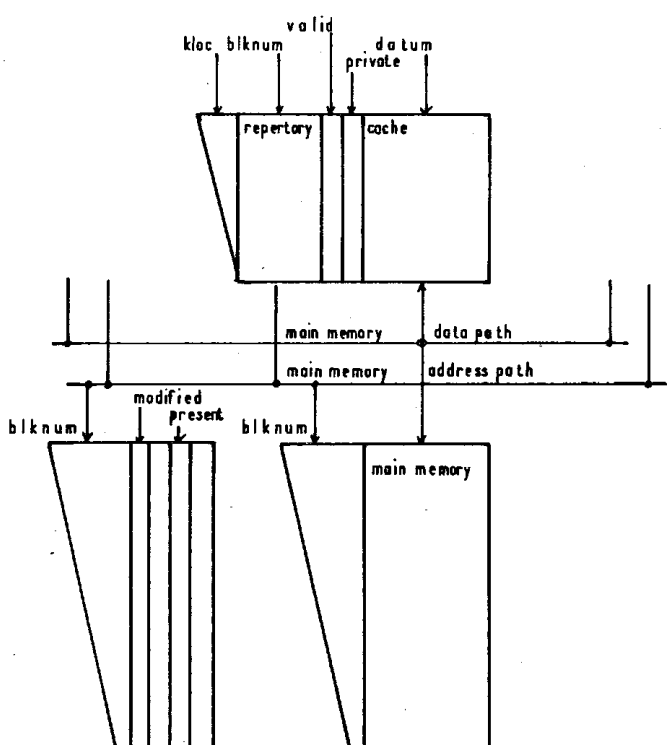

Fig. 1. Conceptual design of a multicache system.

A first filter is implemented by associating a PRIVATE flag to each cache block. When this flag is set in a cache, this cache is the only one to have a valid copy of the block. Hence, all invalidation requests on subsequent STORE accesses may be suppressed (see Fig. 1).

A second filter may be implemented in the main memory control. In the basic design, one associates as much PRESENT flags per main memory block as there are caches in the system; the setting of the PRESENT flag for block a and cache k indicates that a has a valid copy in k. This copy may or may not be identical to its main memory counterpart. When a cache interrogates main memory, invalidation or update requests need be sent only to those cache for which PRESENT is set.

A last filter is obtained by associating a MODIFIED flag to each block in main memory, this flag being reset if the content of a main memory block is identical to all its cache copies. This allows suppression of all update requests on read-only data (e.g., instructions).

The MODIFIED and PRESENT flags are invisible from the processor point of view. They may be stored in a small auxiliary memory which is part of the main memory control. This table is addressed by the high order bits (or block number) of the address, in parallel with the main memory data stacks.

The bit overhead of this scheme is not prohibitive: for moderately sized blocks, it is much lower than the overhead of most error detection/correction designs. If this is felt to be too much, one may divide main memory in fixed size pages and effectively OR together the PRESENT flags of all blocks in a page, at the cost of a slight increase in the number of invalidation requests.

The following properties of the PRIVATE, PRESENT, and MODIFIED flags are essential to the correctness of the coherence algorithm:

1) If PRESENT is set in main memory for block a and cache k, then a has a valid copy in k.

2) If MODIFIED is set in main memory for block a, then a has a valid copy in some cache and has been modified in it since the latest update of main memory.

3) If PRIVATE is set in cache k for a VALID block a, then there is no copy of a in other caches. This implies that there is exactly one PRESENT flag set for a in main memory.

4) If PRIVATE is reset in cache k for a VALID block a, then the contents of a are identical to its counterpart in main memory. This implies that MODIFIED is reset for a.

The data access algorithms must be defined in such a way that these properties are always true, transition times being excepted.

These algorithms are divided in two processes running asynchronously in the cache controllers and in the main memory logic. These two processes exchange commands and synchronization signals. A list of these commands with a short description is given below. The precise description of each command is given in the Appendix as an Algol procedure. Integer arrays are used to represent the main memory, cache memory, and directory. Boolean arrays represent the various flags (Fig. 2).

In this mode of representation, there is no possibility to exhibit the parallelism between the different steps of the

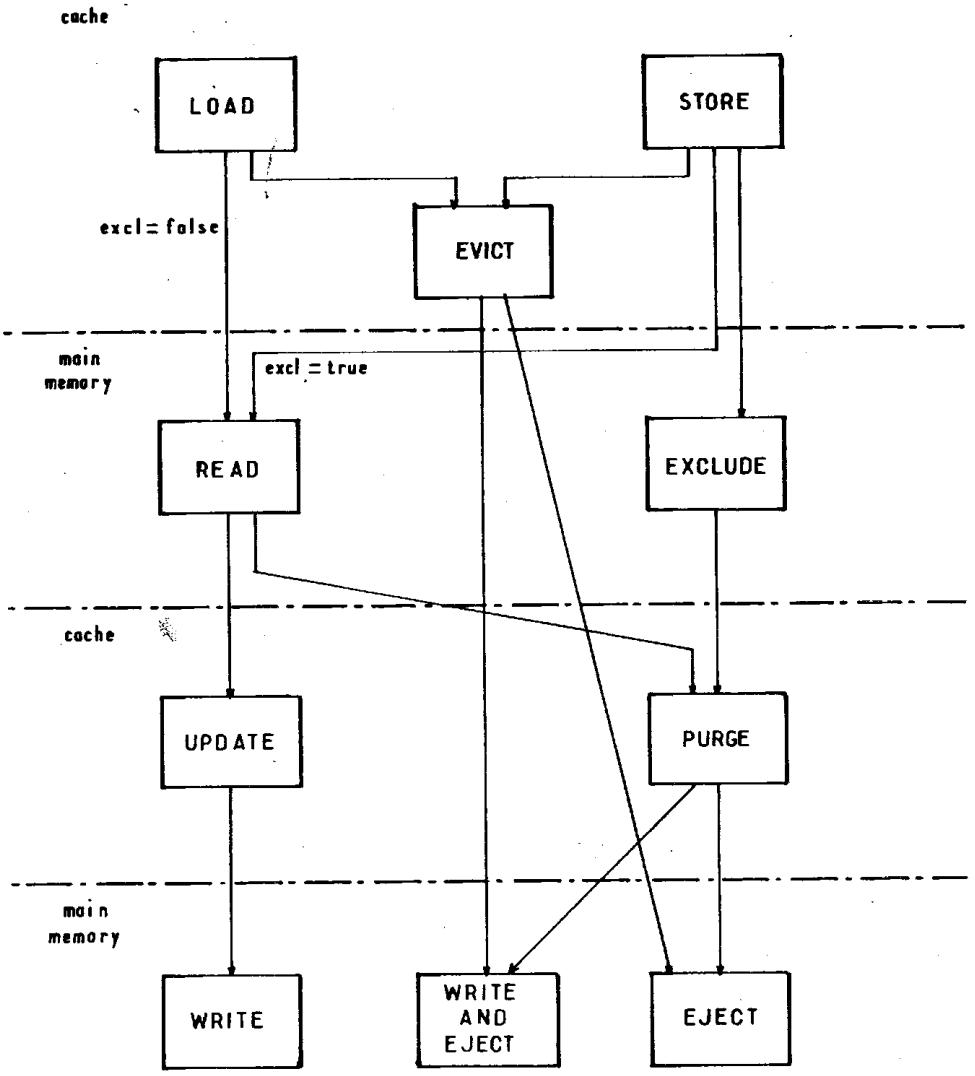

Fig. 2. Structural chart of the coherence algorithm.

algorithms. One may say that each Algol procedure call will be replaced in the real system by the emission of a command, followed by a wait for a DONE signal. To avoid deadlocks, some priority scheme must be devised. Our proposal is to have the cache controllers always obeying the main memory commands, even while waiting for a completion signal. The responsibility for avoiding deadlocks then falls to the main memory control.

A. Instruction Processor Commands

1) LOAD requests the contents of a specified memory location.

2) STORE requests a modification to the contents of a specified memory location.

These commands are executed by the cache controller. For simplicity it will be supposed that LOAD and STORE always act on a full cache block. The modifications for implementing partial LOAD and STORE are self-evident.

B. Cache Commands

1) READ requests the contents of a specified main memory location.

2) WRITE requests a modification to the contents of a specified main memory location.

3) EJECT indicates that a non PRIVATE block has been invalidated in a cache.

4) WRITE AND EJECT combines the effects of WRITE and EJECT.

5) EXCLUDE indicates that a VALID block is going to be modified and that all copies must be PURGED.

READ may be executed in two modes:

a) The standard mode is used following a LOAD. Caches having a copy of the addressed block will be requested to send its contents back to main memory.

b) The exclusive mode is used following a STORE. Caches having a copy of the addressed block will be requested to invalidate it.

In the Algol procedure given below, these two modes are distinguished by a Boolean argument.

These commands are emitted by the cache controller and executed by the main memory controller.

C. Main Memory Commands

1) UPDATE requests that the contents of the addressed block should be copied back to main memory.

2) PURGE is similar to UPDATE, but the addressed block must be invalidated.

These commands are emitted by the main memory controller and executed by the cache controller.

D. Variations on the Presence Flag Technique

It may be felt that associating several presence flags to each block in main memory is too much of an overhead. A solution is then to divide the main memory in pages (which may or may not be the same as the pages used for a demand paging algorithm, if any), and to associate a presence flag to a page. The setting of such a flag will indicate that at least one block was copied from the page into the cache. To help in the setting of these flags, one must implement block counts, which are conveniently located in each cache in an auxiliary low speed associative memory. The block count for a page is incremented at each READ and decremented after a replacement or a successful EXCLUDE. When it steps down to zero, WRITE is replaced by WRITE AND EJECT, thus resetting the corresponding presence flag in main memory.

There are other interesting variations on the basic technique. For instance, when it is found after a LOAD or STORE that the latest version of a block is not in main memory but in a cache, it is possible to transmit directly its contents from cache to cache. The main memory may be updated in parallel with this transmission. It is also possible to postpone this operation until the block is evicted from all caches.

Other variations aim at a reduction of the miss ratio and are beyond the scope of this paper. Examples of those will be found in Bell *et al.* [1]. Another example is the "partial store through" mode in which a STORE is executed in the cache if the block is present and in main memory if absent. The reader will easily convince himself that these variations have no effect on coherence.

E. Performance Estimates

A rough comparison of different coherence schemes may be given under simplifying hypotheses on the data access behavior of the system. As a performance index, we will use the ratio of overhead cycles (execution of PURGE and UPDATE) to useful cycles (LOAD and STORE).

The number of caches in the system will be n. Each will contain k blocks, while the main memory will contain m blocks. α, β, and γ will designate, respectively, the proportion of instruction and constant fetches, of variable fetches, and of modifications. Obviously,

$$\alpha + \beta + \gamma = 1.$$

We will suppose that the contents of each cache are a random selection of the contents of main memory. Hence, the probability for a given block to be in a given cache will be k/m. The data accesses of a processor will not be equally distributed in memory. A proportion $(1 - \varepsilon)$ will be found in the associated cache, while the remaining accesses will be randomly distributed in memory. For the sake of simplicity, we will assume that the hit ratio is the same for LOAD and STORE accesses. This is not exactly true in reality.

The overhead ratio of the classical solution is simply

$$\rho_1 = (n-1)\gamma$$

(each data modification will induce a PURGE in all other caches).

In the presence flag solution, a first cause of overhead will be the necessity to execute the EXCLUDE command when attempting to modify a block for which PRIVATE has been reset. Let K be a cache which receives a STORE command for block a which is present. EXCLUDE will be executed in two cases:

1) When another cache K' has executed a LOAD on a since the last STORE to a in K;

2) When a is present in K as the result of a LOAD, and no STORE has been executed on a.

As to the first cause of overhead, it is easy to see that, under our hypothesis, the mean number of cache cycles between two accesses to a in K will be of the order of k. The probability of a LOAD of a by another cache in this time interval will be $(n-1)(\varepsilon/m)$ and the number of exclude per STORE will be of the order of $(n-1)\varepsilon(k/m)$. As k/m is small, this is negligible with respect to $\gamma\varepsilon$ which is the number of exclusive READ induced by STORE in case of a miss.

As to the second cause of additional overhead, an upper bound may be obtained by assuming that each variable block which is brought in the cache will subsequently be modified by a STORE (this is a pessimistic assumption). Let π be the mean number of presence flags which are set for a given block, excluding the one for the cache under observation. The mean number of overhead cycles induced by LOAD on variables (in case of a miss) and STORE (in case of a hit) will be bounded by $2\varepsilon\beta\pi$. The overhead induced by STORE in case of miss will be $\varepsilon\gamma\pi$. Finally, a constant or instruction LOAD will induce no overhead, as the MODIFIED flag will always be reset in this case.

The probability of a block to be present in p caches out of $(n-1)$ will be

$$\binom{n-1}{p}\left(\frac{k}{m}\right)^p\left(1-\frac{k}{m}\right)^{n-1-p}$$

Hence π is given by

$$\pi = \sum_{p=1}^{n-1} p\binom{n-1}{p}\left(\frac{k}{m}\right)^p\left(1-\frac{k}{m}\right)^{n-1-p} = (n-1)\frac{k}{m}.$$

The total overhead is then less than

$$\rho_2 \leq \varepsilon(2\beta+\gamma)(n-1)\frac{k}{m},$$

and the ratio ρ_2/ρ_1 is bounded by

$$\frac{\rho_2}{\rho_1} \leq \varepsilon\frac{2\beta+\gamma}{\gamma}\frac{k}{m}.$$

Typical values are $\beta = 0.3$, $\gamma = 0.2$, $\varepsilon = 0.1$, $k/m = 32\ 10^3/4\ 10^6 = 8\ 10^{-3}$. This gives $\rho_2/\rho_1 \leq 3\ 10^{-3}$, an improvement of nearly three orders of magnitude.

IV. Conclusion

While the work reported here was done without knowledge of Tang [10] the resulting design is seen to be very similar. In particular, allowing for differences in vocabulary, the set of commands exchanged between cache and main

memory are nearly identical. However, Tang clearly believes that the implementation of his scheme requires duplication of all cache directories in the main memory controller. Our main contribution is to show that this is not necessary. One needs only a few bits per block in main memory (or even less if the suggestion in Section III-D is followed). It is likely that the main extra hardware in our design will be needed in the memory data path which must now be able to accommodate a bidirectional control flow.

The presence flag technique was described in the familiar context of a multicache, multiprocessor configuration. It is, however, applicable in a whole range of different situations.

A first case is that of I/O processors. It is possible to connect them directly to a main memory as long as they conform to the rules set in the preceding paragraphs, in particular as to the use of EXCLUDE and EJECT commands. The technique is specially well adapted for the exchange of orders and status information between the I/O and instruction processors.

The LOAD AND SET instruction will be implemented as an indivisible LOAD-STORE cycle from the processor to the cache; there will be no need to interlock access to the main memory. The reader may wish to convince himself that a processor executing a "busy waiting" loop (LOAD AND SET followed by a test) will not access the main memory unless another processor modifies the block addressed by the LOAD and SET instruction.

Another case is that of extracts from pages and segment tables which are held in a fast associative memory to expedite the conversion between virtual and real addresses. This memory may be considered as a kind of cache. Use of presence flags will obviate the need for systematic invalidation of the associative memory at task switching time.

Last but not least, these mechanisms may be used at all levels of a storage hierarchy, as long as a level is split in separate units which are not equally accessible from all processors.

APPENDIX

PROGRAM GLOSSARY

search	an unspecified Boolean procedure that inspects the repertory for a given block address. In case of a hit, "search" returns the cache location of the block and has the value "true."
select	an unspecified integer procedure that selects the cache block to be evicted to make room for a requested block after a miss.
blknum	a main memory block address.
knum	a cache identifier.
kloc	a cache block address.
nk	the number of caches in the system.

CACHE REPRESENTATION

cache [i, j]	an integer array representing the data part of cache j.
directory [i, j]	an integer array representing the main memory address of the block whose contents are in cache [i, j].
valid [i, j]	Boolean arrays representing the VALID.
private [i, j]	and PRIVATE flags.

MAIN MEMORY REPRESENTATION

mainmemory [i]	an integer array representing the data part of main memory.
present [i, j]	Boolean arrays representing the PRESENT and MODIFIED flags.
modified [i]	

```
Integer procedure load (blknum, knum);
      integer blknum, knum;
comment this procedure is executed by the cache in answer
        to a data request by an instruction processor;
begin integer kloc; boolean t;
      t := search (blknum, knum, kloc);
      if t then t := valid [kloc, knum];
      if not t then
      begin kloc := evict (blknum, knum);
            cache [kloc, knum] := read (blknum, knum,
            false);
            valid [kloc, knum] := true;
            private [kloc, knum] := false;
            directory [kloc, knum] := blknum;
      end;
    load := cache [kloc, knum];
end load;
procedure store (blknum, knum, datum);
        integer blknum, knum, datum;
comment this procedure is executed by the cache in answer
        to a data modification request by an instruction
        processor;
begin integer kloc; boolean t;
      t := search (blknum, knum, kloc);
      if t then t := valid [kloc, knum];
      if not t then begin kloc := evict (blknum, knum);
                    cache [kloc, knum] := read (blknum,
                    knum, true);
                    valid [kloc, knum] := true;
                    directory [kloc, knum] := blknum;
                    end;
                else if not private [kloc, knum]
                    then exclude (blknum, knum);
      private [kloc, knum] := true;
      cache [kloc, knum] := datum;
end store;
integer procedure read (blknum, knum, excl);
        integer blknum, knum; boolean excl;
comment this command is executed by main memory in
        answer to a data request by a cache;
begin integer i;
      for i := 1 step 1 until nk do
        if i ≠ knum and present [blknum, i]
        then begin if excl then purge (blknum, i);
                   else if modified [blknum] then
                   update (blknum, i);
             end;
    present [blknum, knum] := true;
    read := mainmemory [blknum];
    modified [blknum] := excl;
end read;
procedure exclude (blknum, knum);
        integer blknum, knum;
```

```
comment this procedure is executed by main memory in
        answer to a privacy request by a cache;
begin integer i;
      for i := 1 step 1 until nk do
        if i ≠ knum and present [blknum, i] then
        purge (blknum, i);
      modified [blknum] := true;
end exclude;
procedure update (blknum, knum);
          integer blknum, knum;
comment this procedure is executed by a cache in order to
        return the latest contents of a block to main
        memory;
begin integer kloc; boolean t;
  t := search (blknum, knum, kloc);
  if t then t := valid [kloc, knum] and private [kloc, knum];
  if t then
begin write (blknum, knum, cache [kloc, knum]);
  private [kloc, knum] := false;
  end;
end update;
procedure purge (blknum, knum);
          integer blknum, knum;
comment this procedure invalidates a bloc in a cache;
begin integer kloc; boolean t;
  t := search (blknum, knum, kloc);
  if t then t := valid [kloc, knum];
  if t then
    begin if private [kloc, knum]
      then write and eject (blknum, knum, cache [kloc,
      knum])
      else eject (blknum, knum);
      valid [kloc, knum] := false;
    end;
end purge;
integer procedure evict (blknum, knum);
        integer blknum, knum;
comment this procedure is not a command, but a common
        part of the LOAD and STORE commands;
begin integer kloc, addr;
  kloc := select (blknum, knum);
  if valid [kloc, knum] then begin
    addr := directory [kloc, knum];
    if private [kloc, knum]
      then write and eject (addr, knum, cache [kloc, knum]);
      else eject (addr, knum);
    end;
    evict := kloc;
end evict;
procedure write (blknum, knum, datum);
          integer blknum, knum, datum;
comment a procedure used to update main memory; main-
        memory [blknum] := datum;
procedure eject (blknum, knum);
          integer blknum, knum;
comment a procedure used to reset a presence flag;
          present [blknum, knum] := false;
procedure writeandeject (blknum, knum, datum);
          integer blknum, knum, datum;
comment a combination of write and eject;
begin mainmemory [blknum] := datum;
      present [blknum, knum] := false;
      modified [blknum] := false
end write and eject;
```


ACKNOWLEDGMENT

The authors wish to thank A. Recoque and other colleagues at CII-Honeywell Bull for their constructive criticisms. The presence Flag Solution is covered by a French Patent 75.12014 assigned to CII-Honeywell Bull.

REFERENCES

[1] J. Bell, D. Casasent, and C. G. Bell, "An investigation of alternative cache organization," *IEEE Trans. Comput.*, vol. C-23, p. 346, Mar. 1974.

[2] L. Bloom, M. Cohen, and S. Porter, "Consideration in the design of a computer with a high logic-to-memory speed ratio," in *Proc. Gigacycles Computing Systems, AIEE, Winter Meeting*, Jan. 1962.

[3] C. J. Conti, D. H. Gibson, and S. H. Pitkowski, "Structural aspects of the system 360/85—General organization," *IBM Syst. J.*, vol. 7, p. 2, 1968.

[4] C. J. Conti, "Concepts for buffer storage," *IEEE Computer Group News*, vol. 2, p. 9, 1969.

[5] J. Fotheringhan, "Dynamic storage allocation in the ATLAS computer, including an automatic use of a backing store," *Comm. ACM*, vol. 4, p. 435, 1961.

[6] D. H. Gibson, "Considerations in block oriented system design," in *AFIPS Proc. SJCC*, vol. 30, p. 75, 1967.

[7] J. S. Liptay, "Structural Aspects of the System 360/85. II The cache," *IBM Syst. J.*, vol. 7, p. 15, 1968.

[8] R. M. Meade, "On memory system design," in *AFIPS FJCC*, vol. 37, p. 33, 1970.

[9] A. Opler, "Dynamic flow of programs and data through hierarchical storage," in *Proc. IFIPS Congress*, vol. 1, p. 273, 1965.

[10] C. K. Tang, "Cache system design in the tightly coupled multiprocessor system," in *AFIPS Proc.*, vol. 45, p. 749, 1976.

[11] J. Von Neumann, A. W. Burks, and H. Goldstine, "Preliminary discussion of the logical design of an electronic computing instrument," in J. Von Neumann, *Collected Works, Vol. V*. Oxford: Pergamon Press, 1963.

[12] M. V. Wilkes, "Slave memories and dynamical storage Allocation," Project MAC-M-164. Cambridge, MA: MIT, 1964.

Lucien M. Censier was born in Paris, France, in 1932. He received the Diplome d'Ingenieur from the Ecole Supérieure d'Electricité, Paris, France, in 1956.

Until 1970, his activity was oriented in research and development in the application of advanced technologies to different memory levels. Between 1970 and 1974, he participated in the design and the realization of a new minicomputer. Since 1974 his activity has been devoted towards computer architecture, memory hierarchies, and intersystem communications. He is currently with CII-Honeywell Bull, Les Clayes-sous-Bois, France.

Paul Feautrier was born in Marseille, France, in 1940. He received the Doctorate degree from Ecole Normale Supérieure, Paris, in 1968.

From 1962 to 1968, he was with the Paris Observatory, where he did research in theoretical astrophysics. Since 1968, he has been Professor at the University Pierre et Marie Curie, Paris, where he is Manager of the campus computing facility. He is currently doing research in theoretical computer science and computer architecture.