Fully Associative Disk Drive Caches

CACHES INCREASE PERFORMANCE

Caches are employed to reduce the average time that processors take to talk to memory locations. Two of the more common applications in computer design are caching main memory and caching magnetic memory devices such as hard disk drives. By reducing the average time the processor spends talking to memory, the effective performance of the processor is improved. A cache is a smaller block of memory that is of higher performance than the memory being cached. This could take the form of fast SRAM used to cache slower (and cheaper) DRAM main memory. Or in the case of disk drives (which are very slow compared to main memory) the cache could take the form of DRAM to cache the disk drive thus giving semiconductor speeds instead of mechanical speeds. Caching the disk drive can lead to greater performance increases in applications that are highly dependent on disk transactions. This category includes many common applications. A cache, as described in this Application Note, will cache all categories of disk requests thus transparently caching the directories, file allocation table (FAT), and files themselves whenever files are accessed.

WHY CACHES WORK

A cache's ability to improve performance is due to the fact that most computer code is both highly sequential and greatly loop-oriented, leading to spatial and temporal locality of reference in the code. This means that code that will be used in the near future is likely to be near the code being used now (spatial locality) and also that code being used now is likely to be used again soon (temporal locality). These statistical characteristics of computer code mean that while a program may require vast amounts of disk memory to hold code and data to allow an application to perform all the tasks required of it, most of the time programs only use small amounts of code and much of the code is used repeatedly. Thus, improvements in performance can be made by only moving small blocks from the disk into higher performance memory.

Since write bandwidth is only about 15% of read bandwidth, disk caches can improve performance even more by write buffering of the data stream. In this function, the disk cache operates as a very large first in, first out (FIFO) buffer so that for writes of blocks of code near the size of the cache, the effective bandwidth of the disk drive is increased to the bandwidth of the cache. Statistically, writes to the disk of data smaller than common cache sizes are very common. By buffering writes to disk, the overhead of the disk seek and latency times can also be amortized.

HOW CACHES WORK

A cache works by moving blocks of code in use into faster memory so that the data are available more quickly. Since the cache has faster access time than the disk, it has a wider bandwidth, which means that more data can be transferred in the same amount of time. Additionally, cache, which is constructed with semiconductor memory, can eliminate some of the fixed-disk overhead such as disk access time, thus increasing performance even more.

CACHE MAPPING

Because the cache is a duplicate of a small amount of a much larger memory space, the cache must be mapped somehow from the disk memory. Several methods are used for this mapping, all of which have advantages and disadvantages. The most widely used methods are based on the principle of associativity and these are further broken down into one-way set-associative (or direct mapped,) n-way set-associative (typically n = 2 or 4) and fully associative.

OTHER CACHE DESIGN CHOICES

Another permutation of cache architecture is whether the cache is designed as look-through or look-aside cache. Look-through cache lies in the data path between the processor and hard disk drive and buffers all information flow going to the disk drive. The data request from the processor "looks through" the cache. The cache decides if the data requested will be supplied to the processor by itself, in which case it never tells the drive of the request, or if the data need to be retrieved from the drive, the cache forwards the request on to the drive. The other choice is look-aside cache where the cache does not buffer the data flow but simply monitors the data requests and data

transfers and copies the data as they pass back and forth on the bus. The main advantage of the look-through cache is the fact that on the memory side of the cache the bus traffic is reduced thus allowing other activity. This is important in designs for main cache, i.e. cache between a processor and main memory, since the activity is on the system bus that squeezes out other system bus activity by bus masters other than the processor. This advantage does not apply to the disk system under consideration since there are no other bus masters. Although look-through cache is more complex than look-aside cache, a look through cache is required if the cache will be modifying control information as it passes from the disk drive to the host.

DEGREE OF ASSOCIATIVITY TIED TO PERFORMANCE

In one-way set associative caches, the main memory can only be mapped into one location of the cache memory. In n-way set-associative caches, any location in main memory can be mapped into n locations in the cache. In a fully associative cache, any location in disk memory can be mapped into any location in the cache. Figure 1 shows how tracks from the disk map into a one-way set-associative cache. In set-associative mapping schemes the drive memory is divided into groups that are the size of the cache memory. These are labeled groups 1 through 4, in the figure, although typically there are more groups.

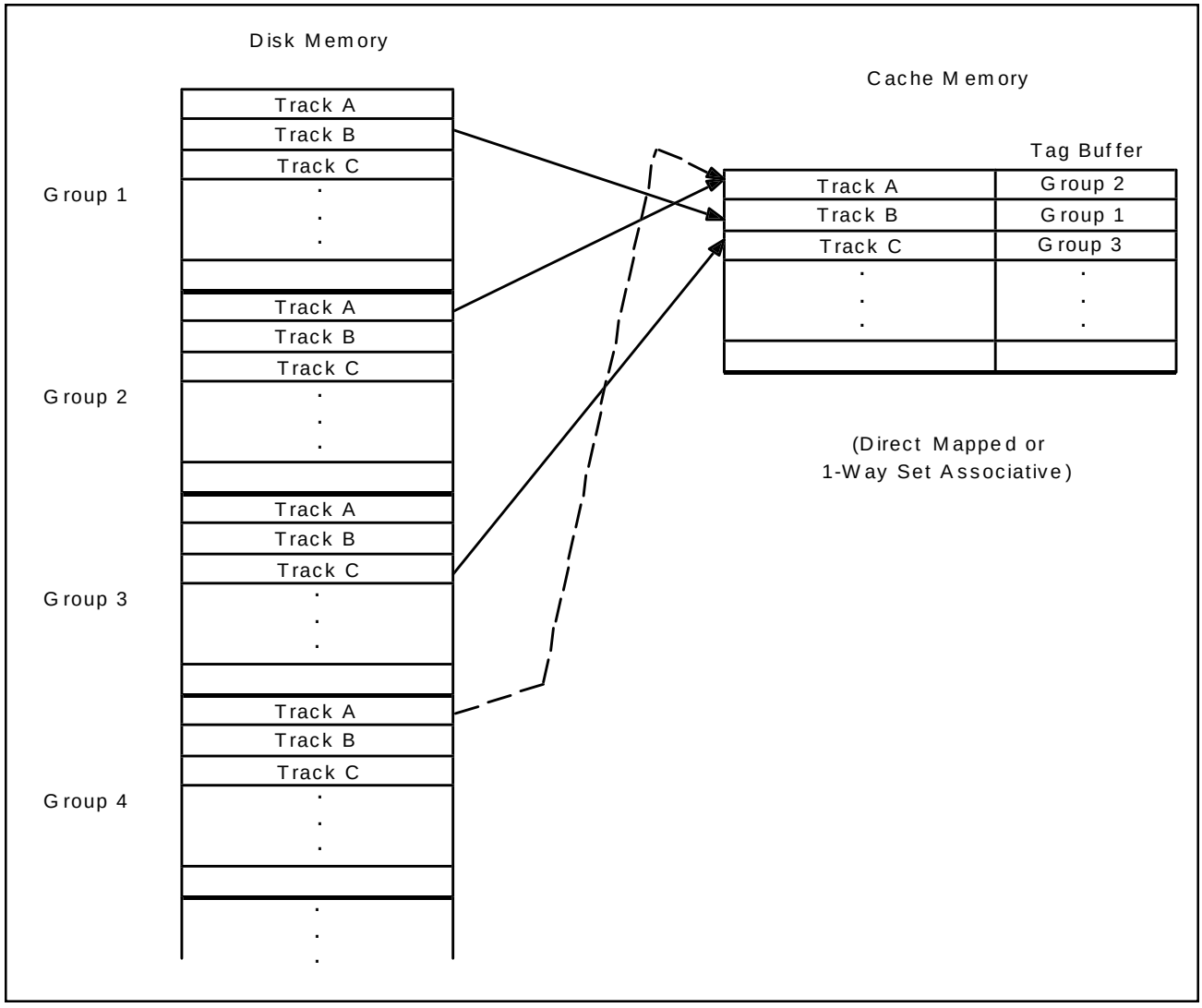

Figure 1: One Way Set-Associative Mapping

Suppose for example that track B from group 1 is written into the cache. Since the cache is direct mapped, track B must go into the place in cache reserved for track B. The cache tag buffer, which is associated with each track, will be written with the offset of the group. Note how the cache can hold tracks from different groups as long as they are not at the same relative position in the group. If two tracks are used that have the same relative place in a group, such as track A of group 4 (noted by the dashed line) then the second occurrence of track A will replace the first one. If tracks with the same relative position are used closely together in code, for example in a loop, then the cache will continually overwrite the track, alternating groups. This is called thrashing, and it greatly degrades the cache performance. To find data stored in the cache, the relative position of a track is calculated (for example "B") and then the tag buffer contents for track B are compared to see if the group offset is correct. If the group offset matches, then the track in the cache is the correct one and can be used.

Figure 2 outlines how a two-way set-associative cache is organized. In this case, there are two possible locations for every track that reduces thrashing (but does not eliminate it). On the other hand, it exacts time and cost penalties because extra logic must compare the two tag buffers to determine which set of the cache contains the proper track A (if any).

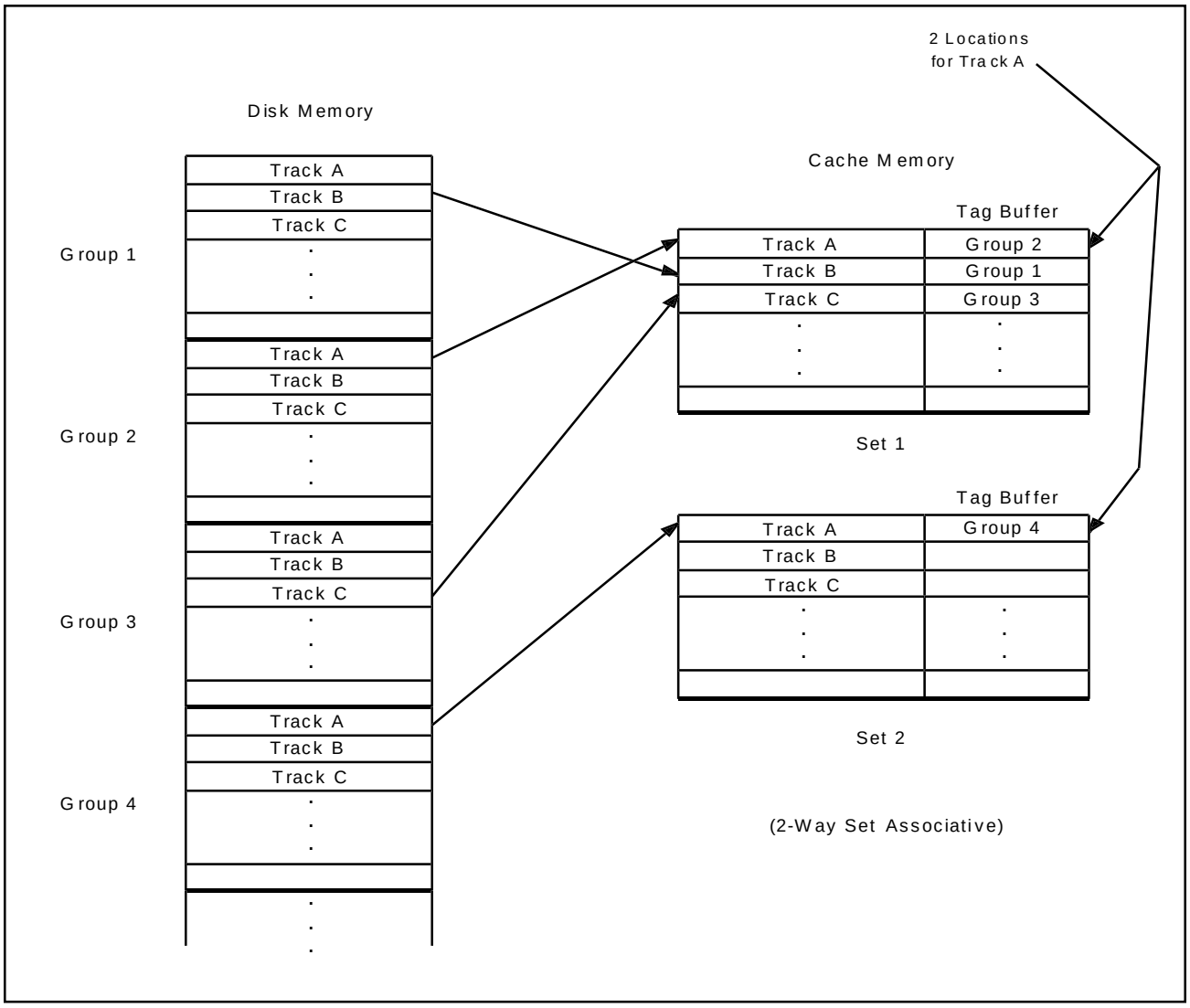

Figure 2: Two-Way Set-Associative Mapping

Figure 3 outlines the construction of a fully associative cache that allows any track to be stored in any location. To do this the usual tag buffer is replaced with a content-addressable tag buffer, then a one cycle search will give the correct track address in the cache (if it exists). Fully associative cache is more flexible too, since the disk memory does not have to be divided up into groups of the size of the cache memory. This leads to a more flexible cache that can accommodate different disk drives when they are updated.

FULLY ASSOCIATIVE CACHE - THE BEST PERFORMER

In the past, the performance advantages of fully associative caches were not realized because of the cost of providing a content-addressable memory (CAM) to perform the searching function. With the availability of MUSIC Semiconductors LANCAM, this increased performance is now available at system costs comparable to current solutions.

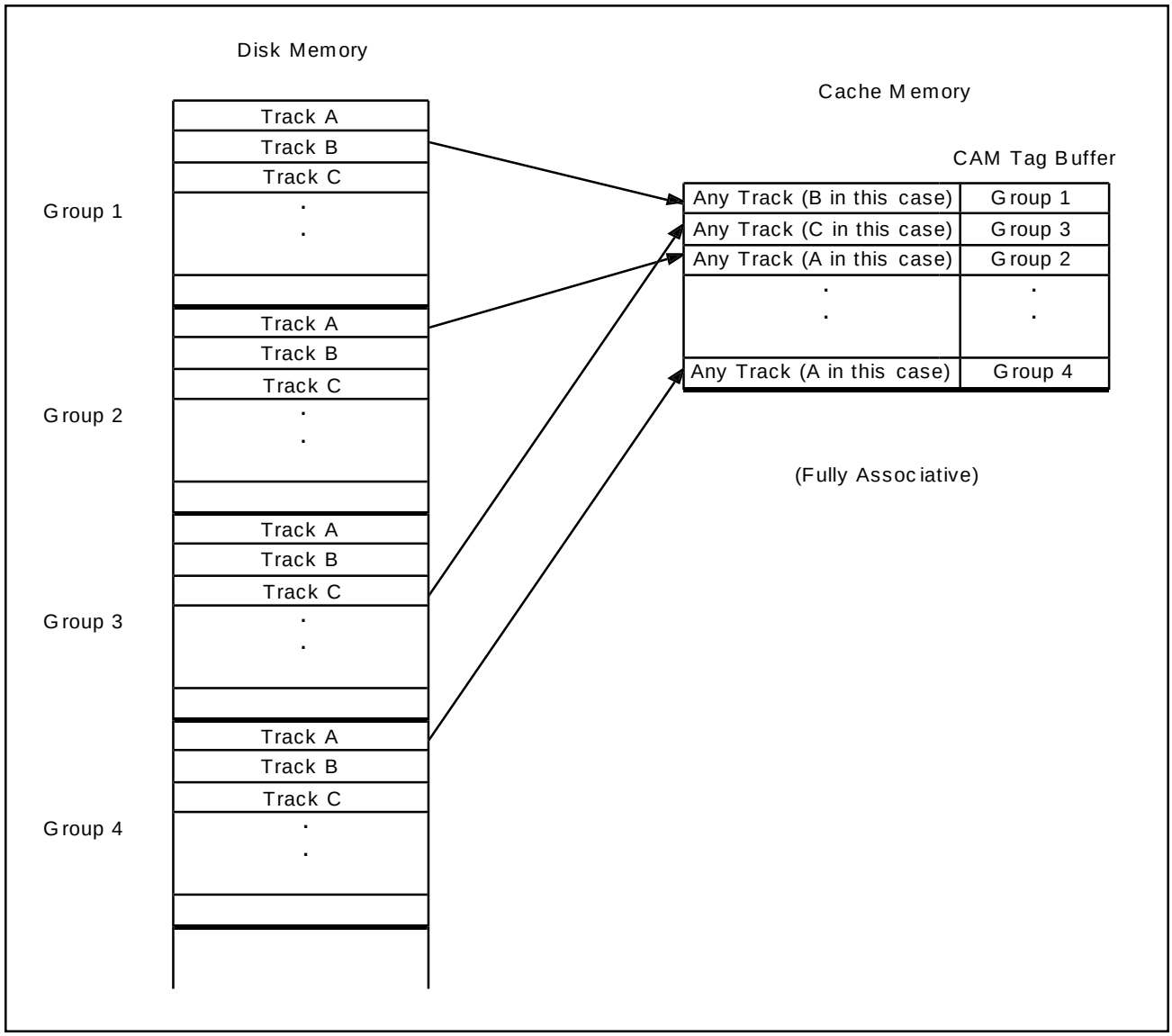

Figure 3: Fully Associative Cache Mapping

AN EXAMPLE SYSTEM

As discussed previously, caches are commonly used to speed up memory access for a processor to main memory or hard disk memory. This Application Note primarily deals with using cache for hard disks, although the concepts are similar.

For purposes of this Application Note, we will consider disk caching for a most common system, a "Compatible PC" using the ISA (Industry Standard Architecture) bus and using an Intel processor. Although this system is possibly the most common hardware in existence it is, nevertheless, a rather complex system in itself. Additionally, the ISA based PC is the foundation for all of the higher performance PC architecture today and it considers only functions (blocks) and interfaces (arrows). When viewed on a systems level however, it is much easier to design since only one function and its interfaces must be considered. Note the top dotted box that encloses a subsystem, the IDE Interface Logic. The first cache design described simply replaces that subsystem with another subsystem that performs all the same functions and talks over the same interfaces.

A PC consists of hardware and software working in concert to accomplish tasks that the user finds useful. Most users would be pleased if they only had to think of the task at hand to accomplish the work, this would be a really "user friendly interface." To eliminate as much work as possible, system designers and software engineers try at every opportunity to hide lower level processes. In the PC, this building of a user friendly interface has occurred over time and built layer on layer, since the builders of the system had to keep older foundations in place to make the current PCs compatible with the large installed base of older PCs. A PC consists of the hardware, such as the processor, and standards used to assemble the hardware, such as the specification of the system bus. Additional components of the system are the BIOS, or basic input/output system, and the operating system kernel, DOS, (or disk operating system). DOS in itself is written like an onion with the interface to the user, the prompt, as the outside layer. To bring more user friendliness to the system, another layer of software, Windows™, was introduced as another layer of the onion to make the user interface more friendly by use of a graphical interface.

The BIOS is the lowest level of software in the PC although it is frequently called firmware since it is contained on a ROM (read only memory). The instructions can not be changed by the operation of the machine so it is not exactly soft, but the ROM can be exchanged with another containing modified instructions, thus the instructions are shares many elements with architectures based on other micrprocessors. Figure 4 gives a system overview of the portion of an ISA computer related to the hard disk drive. This system view hides much of the hardware by firm. The BIOS provides the first level of interface between the hardware and higher level software. The BIOS is designed to hide the hardware from the upper software by implementing identical services for identical commands, called interrupts, for the machines most basic functions.

The DOS kernel, which is the next layer of the onion, also implements various hardware-independent services for use by higher software. The services of interest to us are those concerning file operation and disk control, although DOS contains a lot more services than these. The earliest versions of DOS laid out a plan for storing files on floppy disks and subsequent versions were modified to handle the enormous storage capability of the early hard disk drives installed on the PC-XT, ten Mbytes. As drive capacities increased, further versions of DOS were modified to allow larger disks to be used.

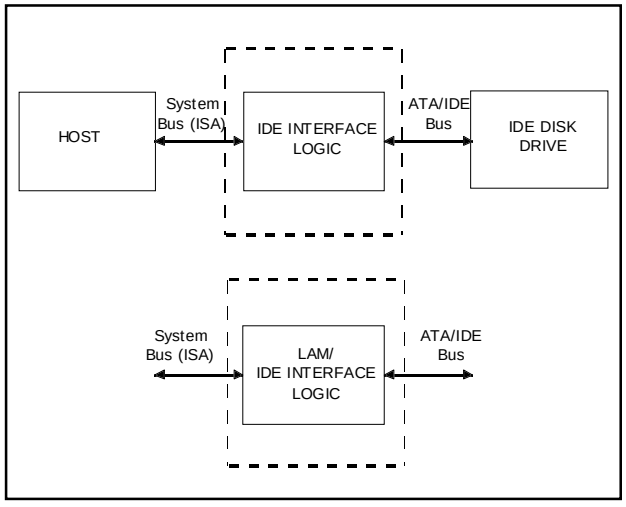

Figure 4: Systems View of a PC

HOW FILES ARE STORED

DOS stores data in files, which are variable sized records of data. These files are organized as a series of bytes (which consist of 8 bits of binary data). DOS stores these files on the disk as a series of clusters of segments. Each segment is 512 bytes long and the clusters consist of one or more segments depending on the type of disk and the version of DOS. DOS keeps track of the files by placing their names in a directory, sort of an index, that is associated with each disk. Also stored in the directory is the location on the disk of the start of the first cluster of a file. Since the file can be much larger than a single cluster, it is stored on a series of clusters. Due to the dynamic nature of a disk, the series of clusters frequently are not contiguous on the disk, so DOS has to have a scheme to find all the clusters for a file. Since the area reserved for a file in the directory is a fixed size and can only contain the address of one cluster (the first one), the location of the other clusters must be stored somewhere else. DOS takes care of this by reserving a location on each disk to keep a record of pointers to clusters, called the FAT (file allocation table). This is the sequence when a request for a file is handed to DOS to service. DOS looks into the directory that returns the location of the first cluster of the file. DOS then looks in the FAT at the location for the first cluster of the file and stored there is the address of the second cluster of the file. Stored in the location of the second cluster of the file is the address of the third cluster of the file, and so forth, until the last FAT entry (pointing to the last cluster of the file) is reached, which has a special reserved address that indicates to DOS that it is at the end of the file. Of course, DOS also retrieves the real data stored in the clusters pointed to by the FAT entries. Storing data like this is called a linked list, since each entry is linked to the next entry. The sequence to retrieve data: get directory entry, get FAT, get file, are to the disk drive the same kind of requests, thus a cache that stores all requests will de-facto cache the directory entry, FAT and file without software or hardware modifications.

SAMPLE DESIGN OF A FULLY ASSOCIATIVE DISK CACHE

A de facto disk drive interface called IDE (coded as ANSI standard X3T9.2791D) predominates in the market for computers based on the ISA bus and derivatives, the so-called PC market. Use of a fully associative cache with drives based on this interface can provide significant performance advantages at minimal cost. The following description is based on the IDE interface, but the concepts can be easily extended to other disk drive interfaces.

In Figure 5, a common implementation of an IDE interface is schematically diagrammed. For disk drive reads or writes, the processor writes the cylinder desired (high and low bytes of the cylinder number), the desired drive and head (1 byte), the starting sector (1 byte), and the total number of sectors (1 byte into 8 bit registers in the disk drive). The drive then reads or writes data to the identified sectors. Additional instructions are available for non-data operations such as formatting, etc.

One method of caching the data from the disk drive is to store an entire track from one side of one platter. Since tracks are defined by mechanical positions of the heads,

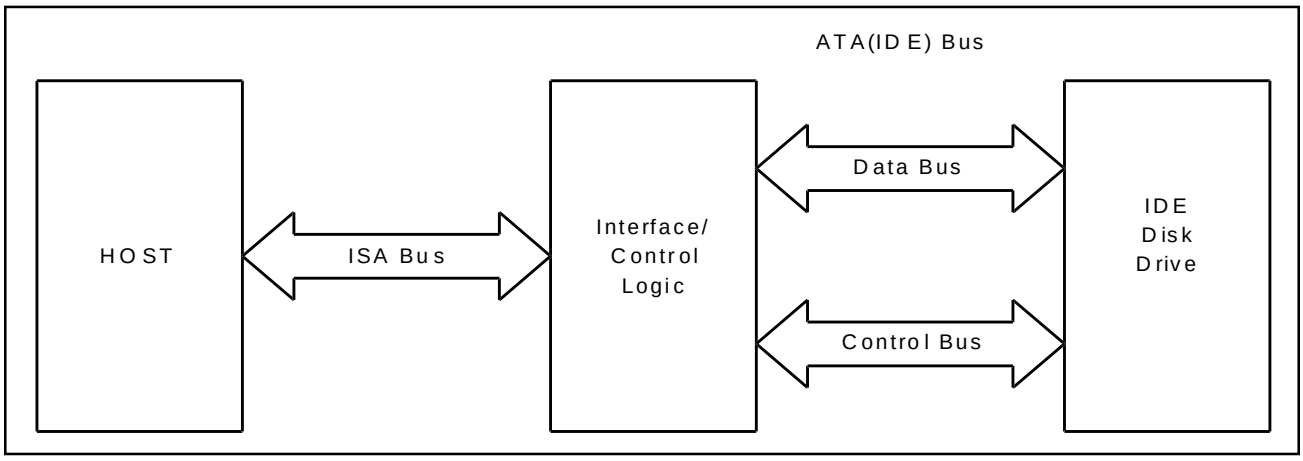

Figure 5: IDE (ATA) Interface

changes in tracks are major contributors to access times. Another major contributor to access times is the latency between when the drive knows what sector it wants to access and when the platters turn to bring that sector under the head, which is on average, half the time required for a full turn of the platters. For illustrative purposes, the drive to be cached will have 17 sectors on each track, each sector having 512 bytes. The total amount of data on a track is therefore 8704 bytes. Any track is uniquely defined by 3 bytes: cylinder high and low, and the Drive/Head byte.

LANCAM REPLACES TAG BUFFER AND SEARCH LOGIC

Referring to Figure 6, the cache is set up using standard DRAM to hold the cached disk data with a fully associative memory, a LAMCAM, to hold the index into the DRAM. The LAMCAM is partitioned into 32 bits of CAM memory, and 32 bits of associated RAM memory. The CAM memory will be written with the 3 bytes that define a track on the disk (which we will call the track name) along with a fourth byte to hold an aging tag. The associated RAM will hold a pointer into the DRAM to the first byte of the first sector of each track stored in DRAM (track pointer). Figure 7 details the partitioning of the CAM memory as used for a tag buffer. The address space of the DRAM is linear, and the addresses (track pointers) to be stored in the associated RAM portion of the LANCAM will be multiples of 8704, the amount of bytes in each track of our disk drive. There are several operating conditions for this cache.

Case 1: Data Read From Disk, Track Not In Cache

The Control logic extracts the cylinder/drive/head 24-bit identifier and presents it to the LANCAM, which in this case indicates a miss (no match). The Control logic then sends the track name (along with the start sector and number of sector bytes) to the disk. While waiting for the disk to return the full track of data, the Control logic first determines if any free memory exists by examination of the full flag. If free memory exists, the Control logic determines where by searching for the next free address. If no free memory exists, then the Control logic frees up memory through a replacement algorithm. After a long while, the disk returns the data. The data are passed to the processor and concurrently written into DRAM memory in the free block. The track name, track pointer, and aging tag are also written to the LAMCAM.

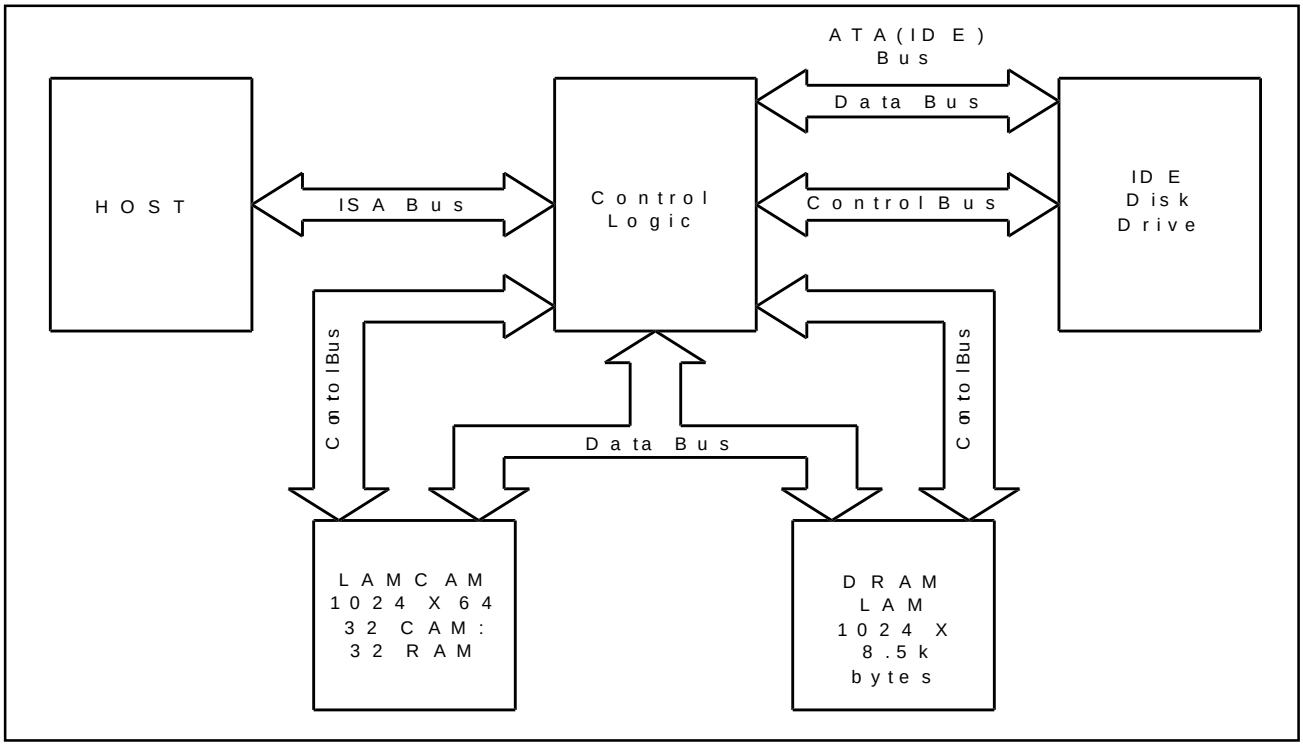

Figure 6: Fully Associative LAMCAM/DRAM Cache

Case 2: Data Read From Disk, Track In Cache

The Control logic presents the track name to the LANCAM to search both "clean" and "dirty" entries, one of which indicates a hit. "Clean" and "dirty" are tags in the LANCAM that indicate that data in the cache match the data on the disk ("clean") or that the data in cache do not match the data on the disk ("dirty"). The Control logic then retrieves the pointer, calculates the offset based on the sectors required, retrieves the data from the proper location in DRAM, and supplies it to the processor. The aging tag can also be updated in the LAMCAM.

Case 3a: Data Write Of A Full Track To Disk, Track Not In Cache

The Control logic writes the data into DRAM and stores the track name and track pointer in the LANCAM setting the validity condition to "dirty" ("dirty" indicates that the cache data have been updated but not yet written to the disk.) The Control logic writes the track data to the disk as fast as disk access allows. With each successful track write to disk, it changes the validity condition to "clean" and updates the aging tag. In this case, the cache acts as a very large FIFO to buffer data. If the data to be written to the disk exceed the capacity of the cache, the Control logic tells the processor to wait.

Case 3b: Case 3b is the same as case 3a, but only some sectors of the track are written to the disk. In this case the new sectors are written to their proper location in DRAM and the LAMCAM track name is tagged "partial." At the same time, the entire track is read from the disk, and when received by the Control logic, the missing sectors are written into the proper place in DRAM to fill out the track. The LANCAM track name tag is changed to "clean."

Case 4: Data Write To Disk, Track In Cache

First use a LANCAM match to check if the data in cache are either "clean" or "dirty." Then write the data over in the old location marking it as "dirty." Write the data to disk as soon as possible and change the tag to "clean."

The design of a cache memory presents some unique challenges. Since data can be stored in two different places, in the cache memory or on the disk drive, or even in both places, care must be taken in the design to ensure that the correct data are supplied to the processor. Additionally,

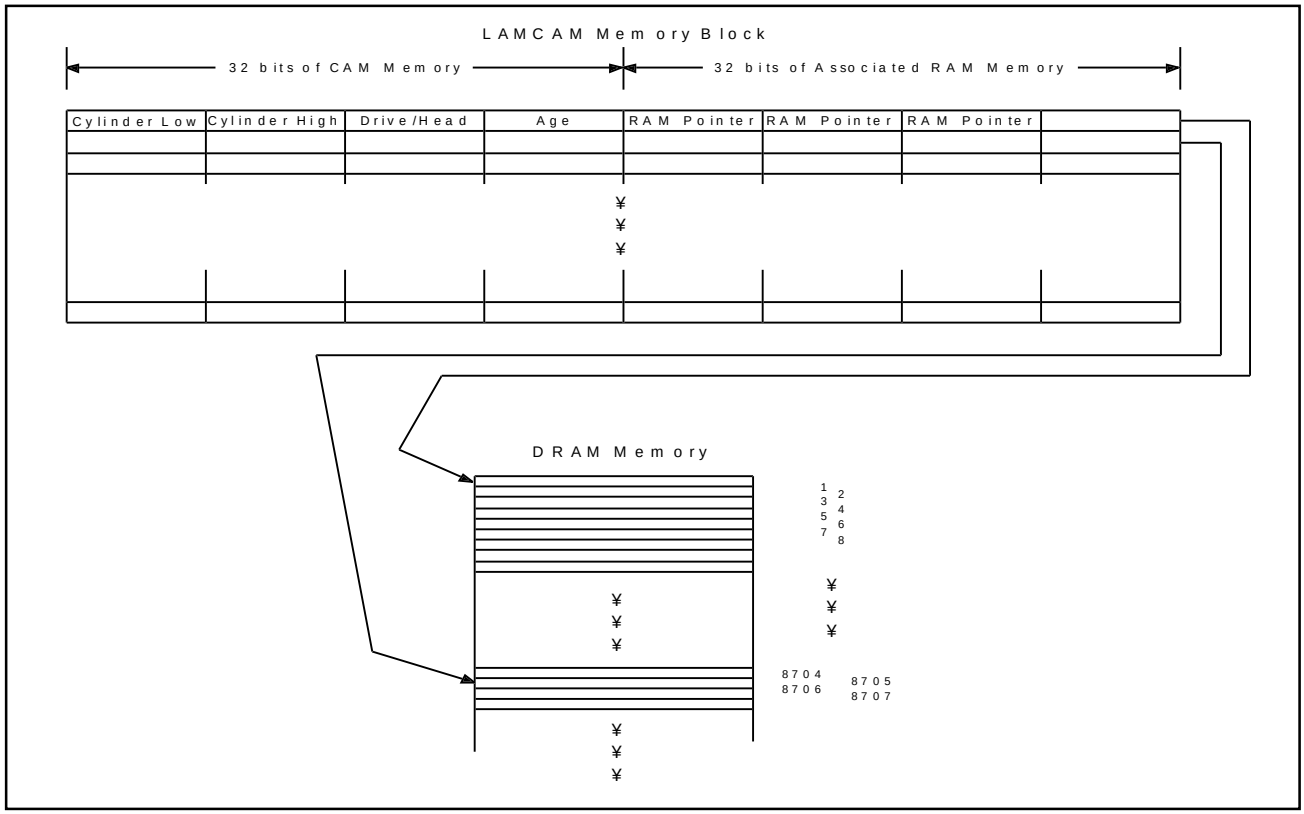

Figure 7: LAMCAM/DRAM Memory Mapping

since the cache is volatile memory (data are lost when the power goes away,) care must also be taken to ensure that the most current data are on the disk drive in a timely manner so that when power is removed from the system, good data are not only stored in the cache and therefore lost. In the following discussion data are referred to as "clean" and "dirty." "Clean" data are data in the cache that match data stored on the disk. "Dirty" data are data in the cache that are not "coherent" with data on the disk. These "dirty" data are, in fact, more recent and more correct than the corresponding disk data. The "dirty" data are generated by the processor modifying "clean" data in the cache. A cache that operates in this manner is called a write back cache, since the data are written first into the cache and then copied from the cache back on to the disk. This problem of processor written cache data not matching the disk data can be eliminated by always writing to the disk directly when data are received from the processor and copying the data into the cache at the same time, (called write through cache). Unfortunately, this advantage carries a disadvantage, which is that now the processor can only write to the cache as fast as it could write to the disk even without the cache. Another scheme to reduce the chance of lost data is to set a time limit for how long "dirty" data can stay in the cache. This is easily implemented by incorporation of a timed "dirty" CAM search along with forced write-back to change all data to "clean." Since high performance is the goal of using a fully associative cache, a cache using the write back design should be considered.

CAM FEATURE ALLOWS EASY AGING ALGORITHM IMPLEMENTATION

Replacement algorithm implementation: Write a sequence number ("age") that increments every four writes into the fourth CAM byte. When the LANCAM is full, change the validity condition of the oldest "clean" tracks to "empty" in units of four. They will be easy to find using a mask register to only search the "age" byte. Since we are only using a byte to store the age in, there are only 256 possible ages. Since the MU9C1480A LANCAM can have 1024 entries, there will be four entries of each age. If more than one LAMCAM is used, the aging algorithm will have to be modified by either increasing the size of the age to more than 8 bits, or increasing the number of entries with the same age to eight (or more) from four.

Control Logic - Figures 8 through 11 are flow charts that detail operation of the Control/interface logic block. Figure 8 starts with decoding the addresses sent to the disk drive over the ISA bus. If the addresses are for disk commands, the logic passes them on to the disk drive without modification. If the addresses indicate that the transaction is for data, then the Control logic determines if the data transaction is a read or write. If the transaction is a data write, then the Control logic first performs a "dirty" search of the LAMCAM and then performs a "clean" search of the LAMCAM. If either of these searches is successful, the data being supplied by the processor are contained in the DRAM cache. Using the starting address (which is determined from the LAMCAM address resulting from the search), the Control logic writes the data to the DRAM cache, marking it as "partial" (if less than a full track) or "dirty" if it is a full track. If the track is not stored in the cache, the Control logic obtains the next free address in the LAMCAM and writes the data in that location marked either "partial," or "dirty," as appropriate. If the tracks stored in the cache are "partial" then the Control logic must request the same track data from the disk drive and fill in the missing sectors in the cache.

Figure 9 details the data read operations of the Control logic/cache memory interface. Figure 10 details the steps necessary to get LAMCAM free addresses. Also included are the steps to obtain free addresses if the cache is full of either "clean" or "dirty" locations. Figure 11 details the steps necessary to clean up the LAMCAM so that all the data are "clean," i.e., the cache data match the disk drive data.

Other functions, not detailed in the series of flow charts, but required of the Control logic, are any required buffering, level shifting, address decoding, or memory housekeeping. A careful analysis of system requirements should be made to determine the feasibility of using a write-back cache. Although the write-back cache can give significant performance advantages, it also presents more opportunities for loss of data. Implementation of a write-through cache requires a slight modification of the Control logic.

A cache constructed as outlined in this Application Note will store the disk data on a track by track basis in a fully associative manner and will be able to start to retrieve the track of data from the cache in little more than the full address/compare cycle time of the LAMCAM (225 nanoseconds for the slowest -12 LANCAM). The speed of data retrieval will depend on the bandwidth of the DRAM memory. The fully associative nature of the cache will lead to higher cache hit rates and more efficiency and performance.

OTHER WAYS TO CACHE DRIVE TO PROCESSOR TRANSFERS

The caching solution previously described, while appropriate for the system described, is not the only choice. A differently designed cache may better serve other architectures and other systems. Using the system description previously presented, consider the following cache design.

When a request is made to DOS for a particular file the following transactions take place. DOS accesses the directory portion of the disk and retrieves the directory entry. This entry points to the location of the first cluster of the file. DOS also must access the disk to retrieve the FAT that has the linked list of clusters for the file. DOS now can determine all the clusters and thus sectors that make up the desired file. DOS now can access the disk again to retrieve the complete file. As can be seen, there is a lot of disk accessing going on to retrieve a file. A cache could be constructed that would store both the directory entries and the linked list of clusters (FAT in this case). Operating systems that read up and store this file information in memory are ripe prospects for improvement through the use of a fully associative cache that would speed up the search for the file clusters. This architecture could result in a separate cache more removed from the disk drive dedicated to list searching. A cache of this sort would be of particular use for systems that typically use lots of files such as network servers.

SELECTING AN ARCHITECTURE

A question that must be answered is how is the cache implemented in the system – indeed, how does the system configuration affect decisions regarding how and what to cache? The system described previously has hardware, software, and performance limitations unique to itself and based in its history. Other systems, such as UNIX®, have different hardware, software, performance limitations, and history. Each architecture must be examined to determine where and how a cache should be used and what the benefits versus costs will be for each application. It is clear however that a fully-associative cache offers higher cache performance than solutions using set-associative architecture.

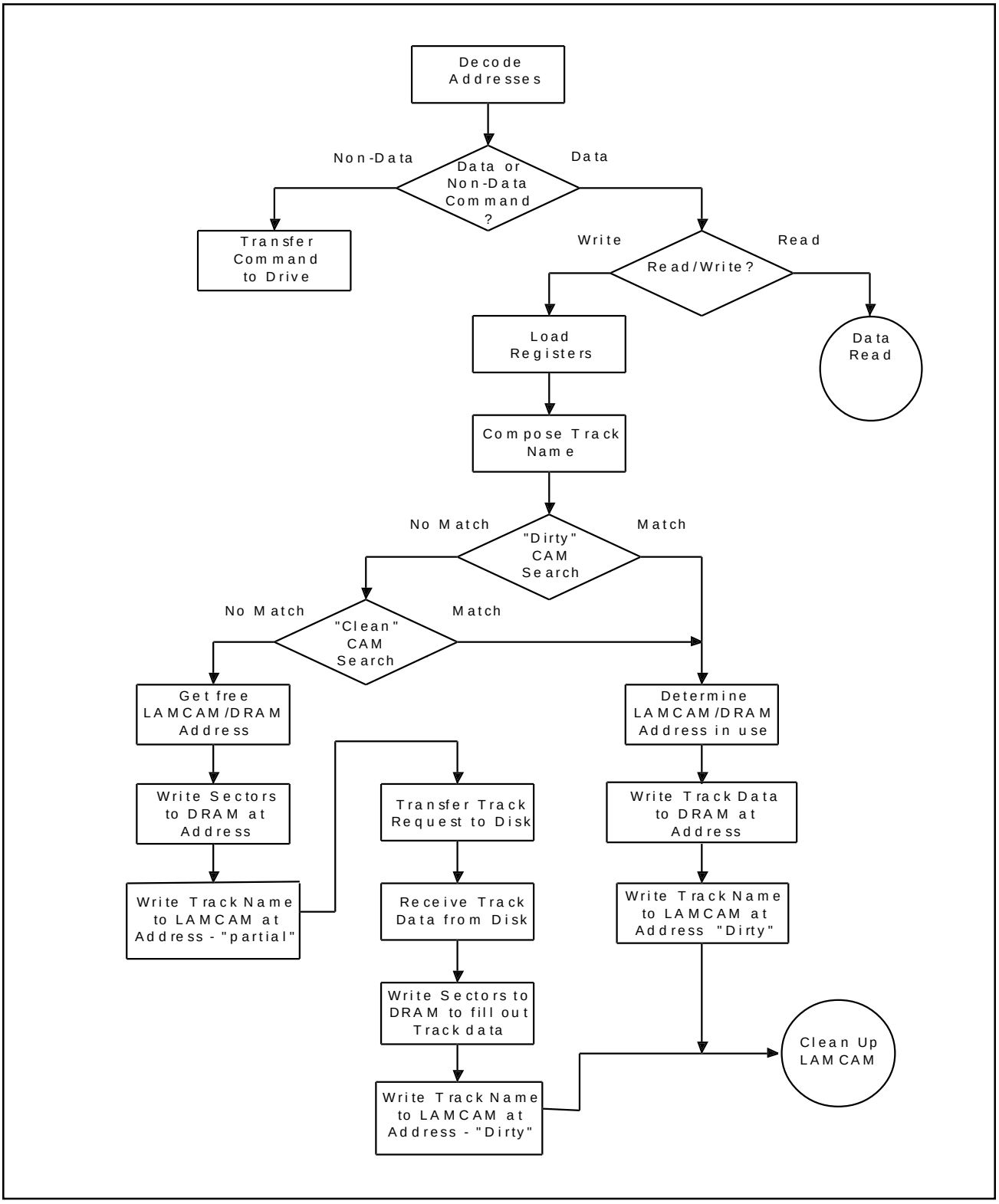

Figure 8: Caching of Data Write Requests to the Disk Drive

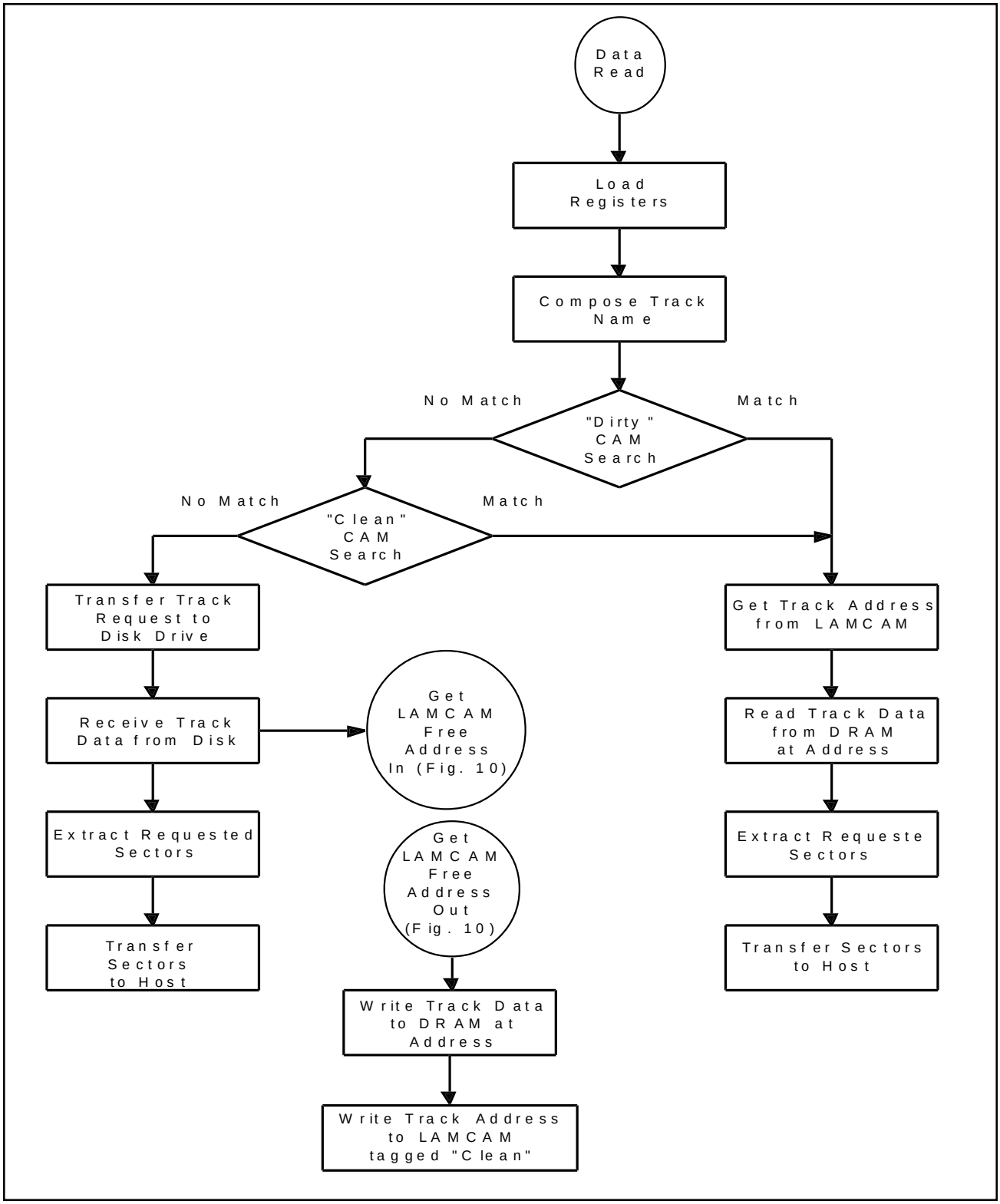

Figure 9: Caching of Data Read Requests to the Disk Drive

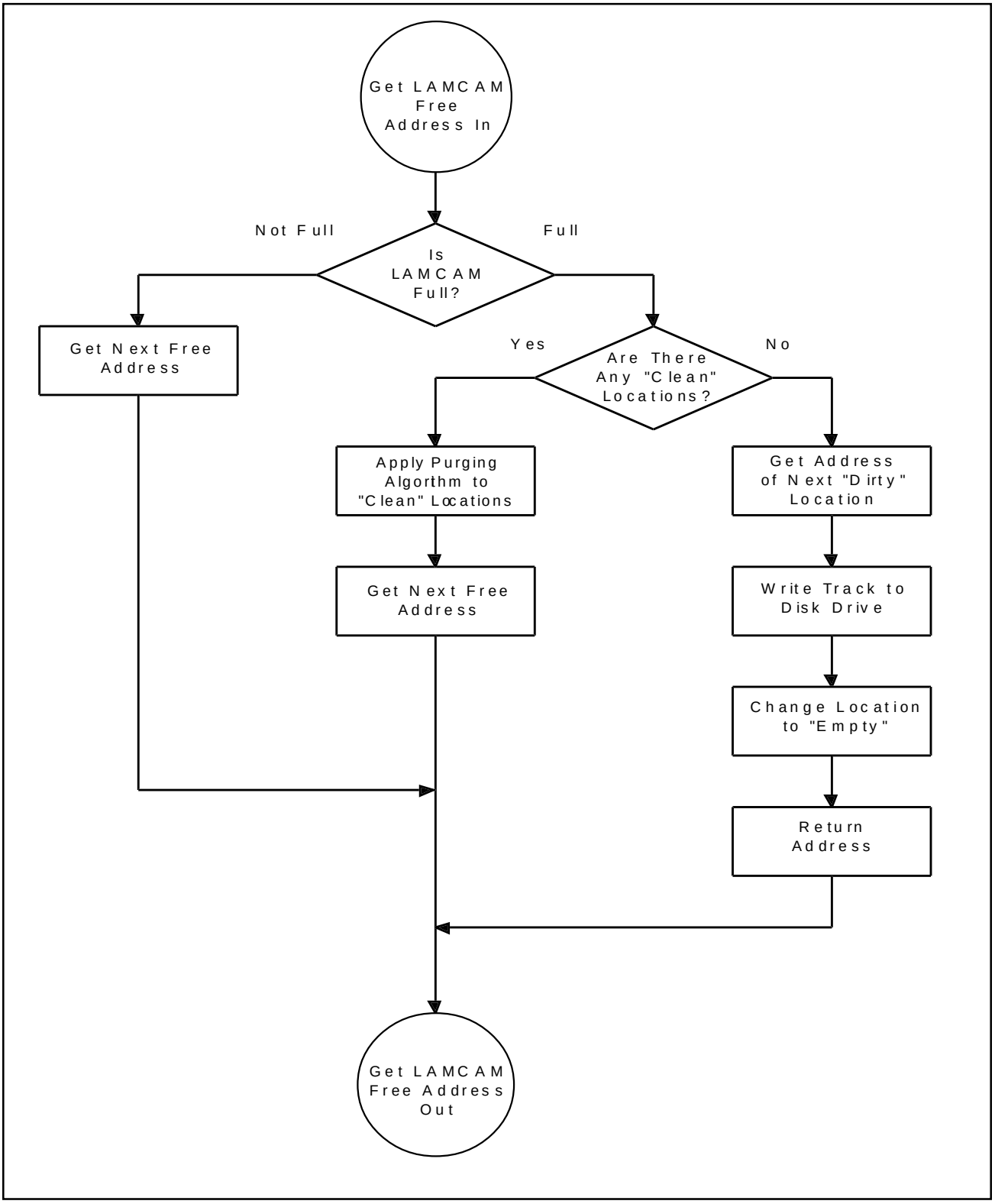

Figure 10: Generation of Free Addresses in the LANCAM Tag Buffer

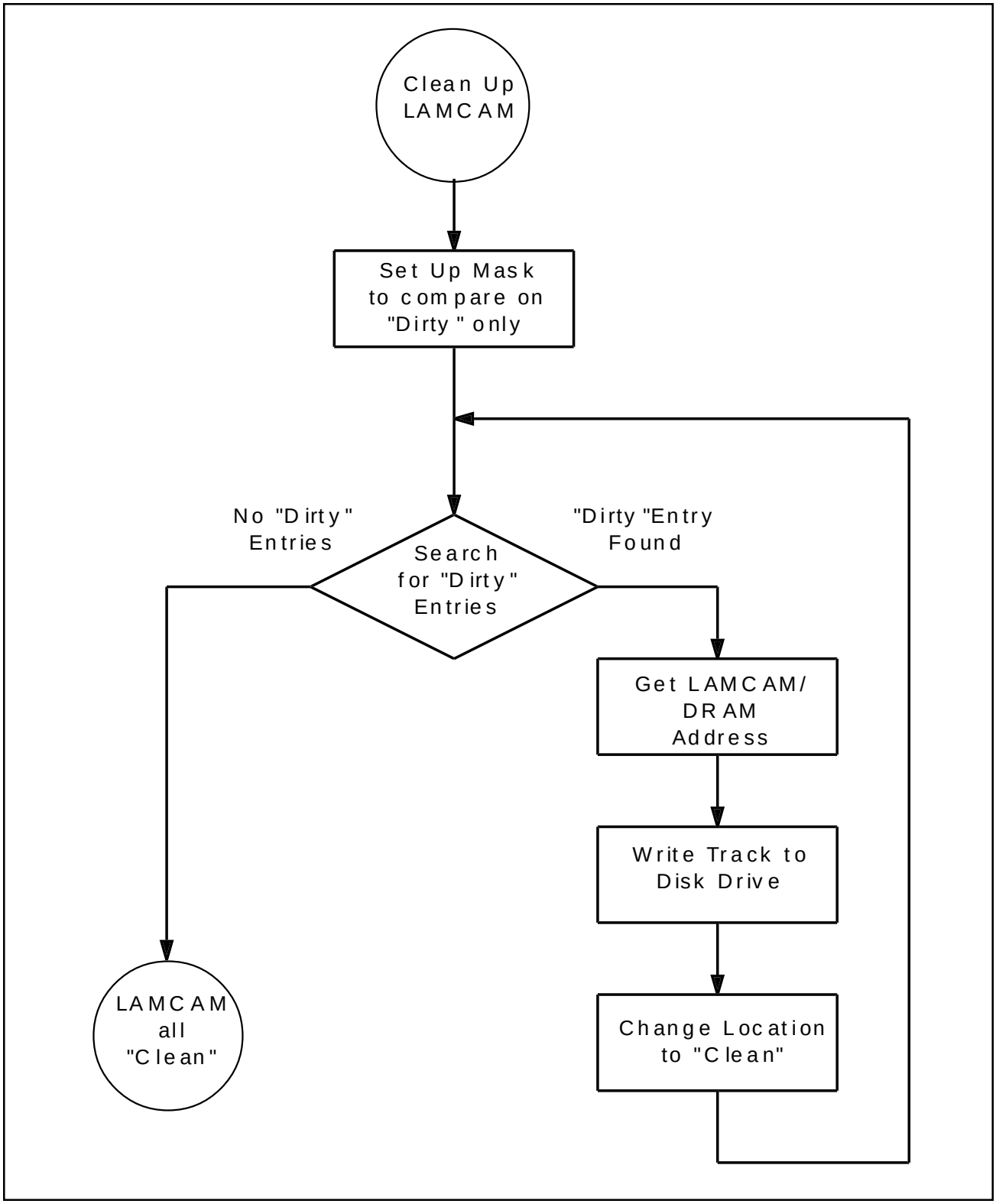

Figure 11: Implementation of Cache Write-Back

NOTES

NOTES

MUSIC Semiconductors Agent or Distributor:

MUSIC SEMICONDUCTORS

http://www.music-ic.com
email: info@music-ic.com

USA Headquarters
MUSIC Semiconductors
254 B Mountain Avenue
Hackettstown, New Jersey 07840
USA
Tel: 908/979-1010
Fax: 908/979-1035
USA Only: 800/933-1550 Tech. Support
888/226-6874 Product Info.

Asian Headquarters
MUSIC Semiconductors
Special Export Processing Zone 1
Carmelray Industrial Park
Canlubang, Calamba, Laguna
Philippines
Tel: +63 49 549 1480
Fax: +63 49 549 1023
Sales Tel/Fax: +632 723 62 15

European Headquarters
MUSIC Semiconductors
Torenstraat 28
6471 JX Eygelshoven
Netherlands
Tel: +31 45 5462177
Fax: +31 45 5463663