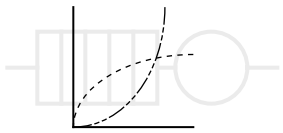

Performance Evaluation

Integrated Systems
Advanced Technology Group

IOTracer Analyzer 1.0 User Manual

Justin Bishop

April 7, 1995

IOTracer Analyzer takes the binary output of IOTracer as input and produces an analysis of the data. The analysis is useful in determining the disk I/O behavior of applications and their interaction with the operating system. The data can be used to tune an application's disk accesses. It also provides a view of an application's interaction with the HFS Disk Cache. The analyzer provides an option for printing a text description of the trace data. When the "trace resource traps" option is used, the text description is helpful in determining the effect an application's resource requests have on disk I/O.

1. Introduction

IOTracer analysis reads the binary trace output created by IOTracer and can either present a summary of file system and driver I/O activity seen in the trace or provide a text representation of the trace.

The summary includes the number of file system and driver read and write calls, the number of bytes or sectors accessed and the time of the calls. A breakdown of the amount of driver time within file system calls is shown. A count of other file system and resource related traps is also included. The text printout shows each IOTracer record traced. The type of record is printed (read/write) and the calls parameters (file/driver number, number of bytes, etc.).

This document gives a description of the analysis and trace printing output of the analyzer. This is followed by a description of how the file system executes read and write requests.

2. The IOTracer Analyzer Application

IOTracer Analyzer is a fat Macintosh application, meaning that it can run on both PowerPC and 68000 Machines. When run on a PowerPC it executes native PowerPC code giving a significant performance gain. It double buffers disk I/O, overlapping the computation operations of one buffer with the I/O operations on the other. It is recommended that the application be run on a PowerPC to take full advantage of both of these features.

IOTracer Analyzer has three menus labeled "File", "Filters" and "Analysis". The "File" menu offers only the "Quit" option, which quits the application. The "Filters" menu contains one menu item, "Timestamps". When this menu item is selected, a dialog box is presented which allows entry of start and stop timestamps. Only records with timestamps greater than or equal to the start value and less than or equal to the stop value will be considered when performing analysis or printing the trace dump.

The "Analysis" menu contains three menu items: "Tab Delimited Analysis", "Text Analysis" and "Trace To Text". the first menu item, "Tab Delimited Analysis" prompts for a trace file (using the Standard File Package) and analyzes the trace, writing its output to file in tab delimited form suitable for spreadsheets. the second menu item, "Text Analysis" analyzes the file and writes its output in a readable text format. The third option "Trace To Text" prompts for a trace filed and prints a text description of each trace record to an output file. The analysis and the text output are shown in the following two sections.

3. Analyzer Output

An example of the analyzer output is shown in appendix A. The output of the analyzer is presented in six sections: information from the trace header, a summary of file level and driver level operations, a decomposition of the file level requests, the number of bytes/sectors requested, a listing of other traps recorded in the trace and eight tables showing the distribution of read and write request sizes.

The trace header information includes the version of IOTracer used to generate the trace, the timer type, whether the spooling option was used and, if it was the time spent spooling and the number of times the buffer overflowed. The header also includes a count of bad records in the trace and timestamps from the time data collection was started and stopped.

The table in the summary section shows the number of read and write calls and the time of the calls. The calls are separated into file level and driver level read and write requests. File level requests are requests issued by applications or system managers such as the Resource Manager for bytes from an open file. A driver request is a request issued by the File Manager (and possibly from other sources) to a device (such as a disk drive).

File level time is the time from the issuing of the request to the return of the request to the caller. Note that for asynchronous requests this may represent the time taken to queue the request and not the time to search the disk cache or access the disk. Synchronous requests do not return to the user until the requested operation has been completed. For synchronous driver calls, driver time is the time from the issuing of the request to the

request's return (the request will not return until the disk operation is complete). For Asynchronous driver calls, driver time is the time from the issuing of the request to the call of the completion routine in the parameter block used in the request. The completion routine is not be called before the disk operation is complete (and disk driver calls always have a completion routine).

The section labeled Decomposition is broken up into five categories: *Sync File Level Reads*, *Async File Level Reads*, *Sync File Level Writes*, *Async File Level Writes* and *Driver Calls Outside Of File Level*. The first four categories show the count and time of the file level calls recorded in the trace and break the time down into components. Figure 1 shows an example of the *Sync File Level Reads* category. The trace contained 600 async file level read requests and the time of the requests was 4,788,269 microseconds. The time is broken up into driver time of 3,216,990 microseconds and non driver time (time executing file system code, etc.) of 1,571,306 microseconds.

Sync File Level Reads

Count:	600	Driver Time:	3,216,990
Time:	4,788,296	Non Driver Time:	1,571,306

Driver Summary	Sync Count	Async Count	Sync Time	Async Time
Driver Reads	158	0	3,216,990	0
Driver Writes	0	0	0	0

Figure 1. File Level Request Time Decomposition

The table in figure 1 shows the types of driver calls that occurred inside the file level call. A driver call is described as being inside a file level call if the driver call is issued (T_{dstart}) after the file level request (T_{fstart}) and completes (T_{dstop}) before the file level call returns (T_{fstop}). Figure 2 shows an example of a driver call inside a file level call. In this example, there were 158 synchronous driver read requests inside the 600 file level read requests.

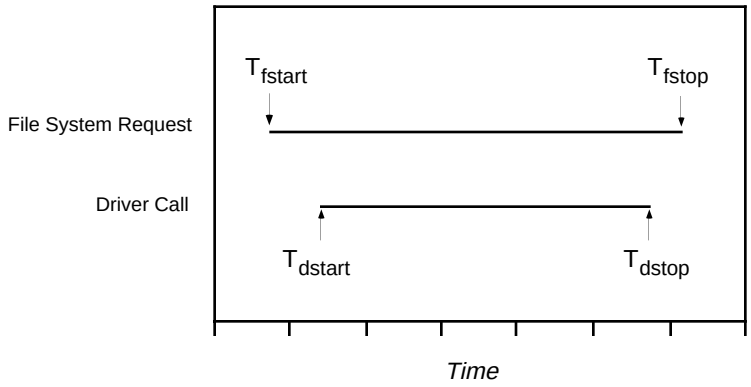

Figure 2. Driver Call Inside of a File Level Call

The fifth category consists of a table labeled "Driver Calls Outside Of File Level". A driver call is considered to be outside of a file level call if there is no interval $[T_{fstart}, T_{fstop}]$. in which T_{dstart} lies. Async file level requests may result in driver calls outside of file level calls (Figure 3 shows an example). Driver calls occurring outside of file level calls can also consist of catalog, extents and volume bitmap reads and writes as well as cache moveouts and flushes.

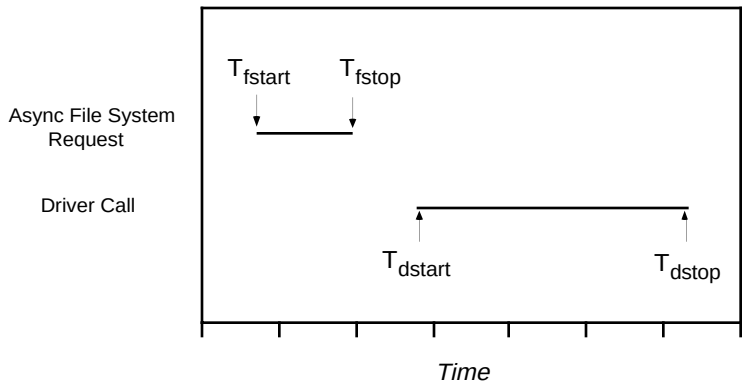

Figure 3. Driver Call Outside of a File Level Call

Note that async driver calls may be issued before the async file level request that generated (i.e. put the request on the file system queue) the call returns and complete after the file level call has returned. Thus the driver time is partially inside and partially outside the file level call (See Figure 4). In this case the driver request is counted as a driver call inside the file level request and $(T_{dstart} - T_{fstart})$ is counted as time inside the file level call and $(T_{dstop} - T_{fstop})$ is counted as time outside of a file level call.

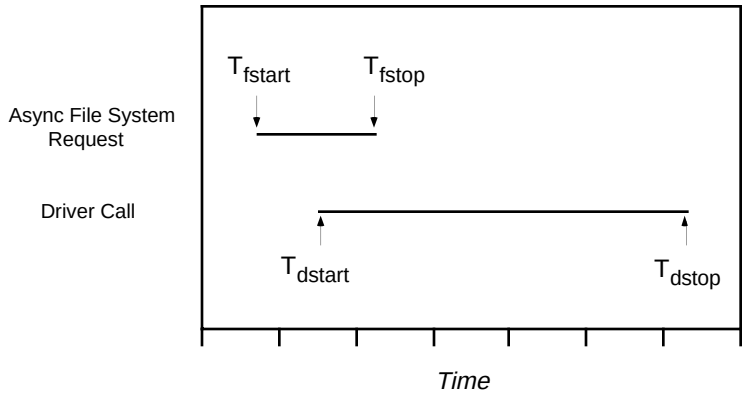

Figure 4. Driver Call Inside and Outside a File Level Call

4. Text Output

An example of the output produced when the "Trace To Text" menu item in the "Analysis" menu is selected is shown in figure 5. The example shows an excerpt from a trace of a launch of the application MacFlow 3.7. Each trace record is printed on its own line starting with the name of the Atrap that generated the record. The name is followed by the start timestamp and stop timestamp (if applicable: some types of record contain no stop time). The information that follows the two timestamps varies by the type of record.

The example in figure 5 starts with an *Open Resource Fork* request for the file MacFlow 3.7. The trap was called at timestamp 3,527,462 and returned at timestamp 3,534,061. The next record, a *Get File Info* call, was called at timestamp 3,532,187. because the T_{start} of the *Get File Info* is less than the T_{stop} of the *Open Resource Fork* Atrap, the *Get File Info* was called before the *Open Resource Fork* call returned. The name of the *Get File Info* trap is indented to show that it occurred within another traced trap.

```
Open Res Fork    3,527,462   3,534,061   MacFlow 3.7
  GetFileInfo    3,532,187               MacFlow 3.7
Read             3,534,195   3,552,318   File: 7E9F  Bytes:   24  IOBuf:  220A20  FP:      0
    Read DRVR    3,534,725   3,552,188   Drvr: FFDF  Bytes:  200  LS:     FE1F3
Read             3,552,563   3,582,447   File: 7E9F  Bytes: 1F6B  IOBuf:  4CB0EC  FP:  55AA3
    Read DRVR    3,552,945   3,562,846   Drvr: FFDF  Bytes:  200  LS:     FE4A0
    Read DRVR    3,563,131   3,576,449   Drvr: FFDF  Bytes:  1E00 LS:     FE4A1
    Read DRVR    3,579,648   3,582,322   Drvr: FFDF  Bytes:  200  LS:     FE4B0
Read             3,583,480   3,598,255   File: 7E9F  Bytes:    4  IOBuf:  1D004BA FP:  401D8
    Read DRVR    3,583,838   3,598,131   Drvr: FFDF  Bytes:  200  LS:     FE3F3
Read             3,598,413   3,598,639   File: 7E9F  Bytes:    A  IOBuf:  220A20  FP:  401DC
CloseResource    3,598,971               File: 7E9F
Close            3,599,755   3,601,353   File: 7E9F
```

Figure 5. Text Output Example

The next record in the example is a file level read (a file level read/write request is printed with the name of the Atrap, a driver level request is printed with the name followed by 'DRVR'). The read request was for the file with File ID 7E9F in hexadecimal (See *Inside Macintosh: Files* page 2-24 for the definition of a File ID). the request was for 24 hex bytes. The I/O buffer for the request was 0x22A20. The file position (or file mark, See *Inside Macintosh: Files* page 1-9 for the definition of a file mark) was 0.

The file level read request is followed by a driver read request. Examination of the timestamps shows the driver call to be inside the file level call, as in figure 2 (this can also be seen by the indentation). The request was issued to the driver with a driver reference number of FFDF hex. The request was for 200 hex bytes (one 512 byte sector). The logical sector requested was FE1F3 hex.

5. How The File System Treats File Level Requests

This section gives a basic view of the logic the Macintosh file system applies to file system read and write requests. File level read requests are generally issued by applications using the FSRead or PBRead (and PBReadAsync) routines. File level writes are generally issued using the FSWrite or PBWrite (and PBWriteAsync) routines. For each open file, the File Manager maintains a current position marker, called the file mark. The file mark is the offset in bytes from the beginning of the file at which the next read or write operation will start.

Sector alignment is an important concept in understanding how the file system responds to requests. If the file mark is evenly divisible by the disk drive's sector size (usually 512 bytes), the request is referred to as *front sector aligned* or *front aligned*. If the file mark is not evenly divisible by the sector size, the request is *front unaligned*. If the byte after the last byte in an operation (the file mark + the request size) is evenly divisible by the sector size the request is said to be *rear sector aligned* or *rear aligned* otherwise it is *rear unaligned.*

It is common to see a file level multi-sector read request split into three driver read requests, one for the first sector, one for the middle sectors, and one for the last sector. This occurs when the request is both front and rear unaligned. Since only the requested bytes can be sent into the application buffer, the first and last sectors must be read from disk into the disk cache (assuming they aren't already there, in which case you have a cache hit). Only the requested bytes from the sector are copied into the application I/O buffer. The middle sectors are read from disk directly into the application I/O buffer (so they can be read in one contiguous operation). In some cases, these middle blocks are copied into the disk cache. Figure 6 shows how a seven sector read request that is front and rear unaligned is treated.

File level requests that span aligned sectors may bypass the cache, in which case the aligned sectors will not be copied from the application I/O buffer into cache blocks. The programmer can explicitly request a read or write bypass the cache by setting bit **5** of the ioPosMode field in the parameter block sent to a PBRead/PBWrite call (See *Inside Macintosh: Files* page 2-89). The HFS Disk Cache also has a built in bypass mechanism. The aligned portion of requests above a threshold size will always bypass the cache. The cache bypass threshold is a function of the current HFS Cache size (i.e. the larger the cache, the larger the bypass threshold, meaning that as the cache size is increased fewer requests will bypass it). Note that bypassing the cache only applies to the aligned portion of a request; unaligned portions of a request which bypasses the cache will still access the cache.

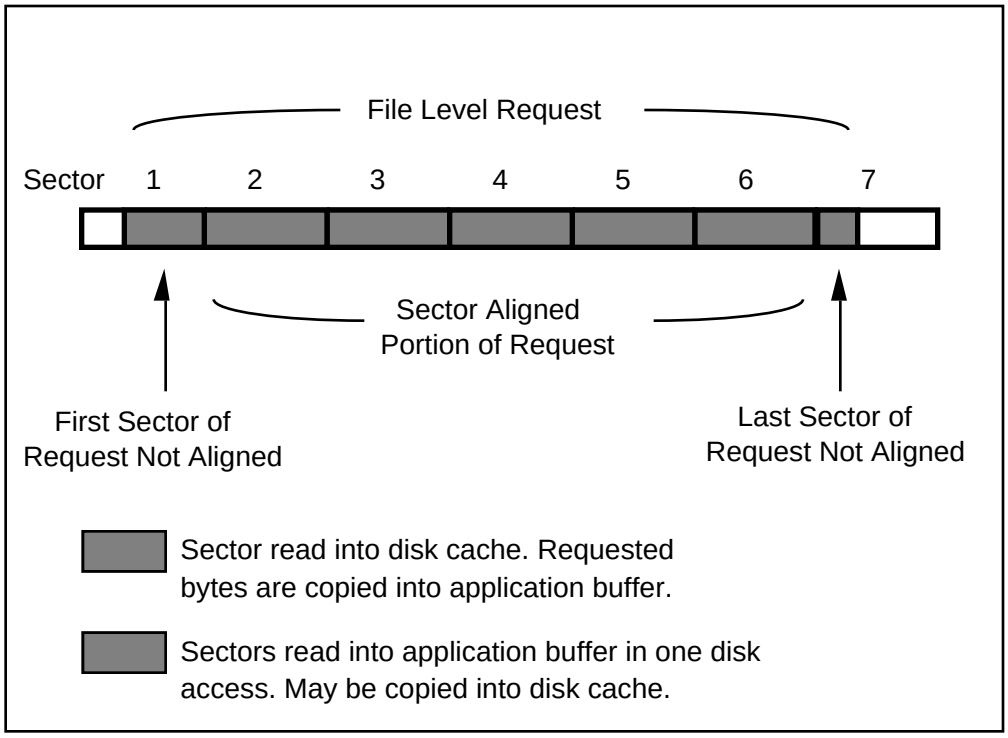

Figure 6. Typical File Read Request

Figure 8 in appendix B shows how the file system treats file level requests. The flowchart is broken up into three sections. Section 1 shows how the first sector of a read request is handled if it is not sector aligned (sector 1 in figure 6). Section 2 shows how the sector aligned portion of the request is handled (sectors 1 to 6 in figure 6). Section 3 shows how the last sector of a request is handled if it is unaligned.

To give an example, because the request in figure 6 is not front aligned, the cache will be searched for the first sector of the request (section 1 of the flowchart). If the sector is found in the cache, no disk read is necessary. If the sector is not found in the cache a free block must be found and the sector read into the cache. If no free blocks are available, the cache replacement must be run to release cache blocks. If dirty cache blocks (blocks that contain data from writes that have not yet been written to disk) are to be released, they are first written to disk. This can cause driver writes to occur within a file level read request.

Next, the request will be checked to see if it bypasses the cache (section 2 of the flowchart). if it does, sectors 2 through 6 will be read from disk into the application buffer (the cache is checked first to ensure that the data on disk is not stale). Note that bypassing the cache only refers to aligned portions of a request, the cache still needs to be accessed for unaligned sectors. If the request does not bypass the cache, the cache will be searched for the sectors. If all sectors are found in the cache no disk read will occur. If any sectors are not found in the cache, a driver read is issued.

In section 3 of the flowchart, the cache will be searched for sector 7 because the request is rear unaligned. If the sector is found in the cache, the request is complete. If not a free block is obtained as in section 1 and a driver read request is issued for sector 7. If the cache replacement algorithm is run to create free blocks, disk writes may occur.

Requests that do not span any aligned sectors will skip section two. Requests of less than one full sector will skip sections two and three.

Figure 9 in appendix A shows the logic for file level write requests. Similar to the file read flowchart, file level write requests are processed in three sections: section 1 for a first unaligned sector (if any), section 2 for the aligned portion of a request (if any) and section 3 for a rear unaligned sector (if any). The logic of file level writes is similar to that of reads. A major difference is that when a write request includes an unaligned sector, the sector must first be read into the cache from disk (causing a driver read call inside a file level write call) and the bytes to be written copied into the cache.

When sectors are copied into cache blocks during writes, the blocks are marked dirty and written at a later time, either during the running of the cache's replacement algorithm or during a cache flush. Similar to file level reads, requests that do not span any aligned sectors skip section two of the flowchart. Requests of less than one full sector will skip section two and three.

File fragmentation can force a multi-sector request to be broken into more than one driver read. Fragmentation can also cause single sector reads of the Catalog and Extents file inside of a file level request.

Appendix A. Analyzer Output Example

```
================================================================================
|                                                                              |
|      IOTracer Analysis of Input File:  IOTracer.out                          |
|                                                                              |
================================================================================
```

Header Info	
IOTracer Version	6.0
Timer Type	Atrap Timer
Spooling	Off
Spool Time	0
Bad Records	0
Overflow Count	0
Start Time	388,033,060
Stop Time	406,961,956
Trace Time	18,928,896

========== Summary ==========

Read/Write Totals	Sync Count	Async Count	Sync Time	Async Time
File Level Reads	600	12	4,788,296	3,515
Driver Reads	238	10	4,134,144	123,127
File Level Writes	10	0	42,015	0
Driver Writes	11	0	85,014	0

========== Decomposition ==========

Sync File Level Reads

Count: 600 Driver Time: 3,216,990
Time: 4,788,296 Non Driver Time: 1,571,306

Driver Summary	Sync Count	Async Count	Sync Time	Async Time
Driver Reads	158	0	3,216,990	0
Driver Writes	0	0	0	0

Async File Level Reads

Count: 12 Driver Time: 1,256
Time: 3,515 Non Driver Time: 2,259

Driver Summary	Sync Count	Async Count	Sync Time	Async Time
Driver Reads	0	2	0	1,256
Driver Writes	0	0	0	0

Sync File Level Writes

Count: 10 Driver Time: 34,023
Time: 42,015 Non Driver Time: 7,992

Driver Summary	Sync Count	Async Count	Sync Time	Async Time
Driver Reads	3	0	34,023	0
Driver Writes	0	0	0	0

Async File Level Writes

Count:	0	Driver Time:	0
Time:	0	Non Driver Time:	0

Driver Summary	Sync Count	Async Count	Sync Time	Async Time
Driver Reads	0	0	0	0
Driver Writes	0	0	0	0

Driver Calls Outside Of File Level

Totals	Count	Time
Sync Driver Reads	77	883,131
Async Driver Reads	8	78,995
Sync Driver Writes	11	85,014
Async Driver Writes	0	0

Byte/Sector Counts

File Level Requests

	Bytes
Sync File Level Reads	408,261
Async File Level Reads	5,888
Sync File Level Writes	758
Async File Level Writes	0

Driver Requests

	Sectors
Sync File Level Reads	868
Async File Level Reads	10
Sync File Level Writes	13
Async File Level Writes	0

Other Data

Trap	Count	Time
Open	4	221,300
Async Open	0	0
Close	10	40,126
Async Close	0	0
Get File Info	34	96,362
Delete	3	n/a
Create	2	n/a
Create Res File	0	n/a
Open Res File	1	n/a
Get Named Resource	0	n/a
Get Resource	0	n/a
Get Ind Resource	0	n/a
Load Resource	0	n/a

Appendix B. File Level Request Flowcharts

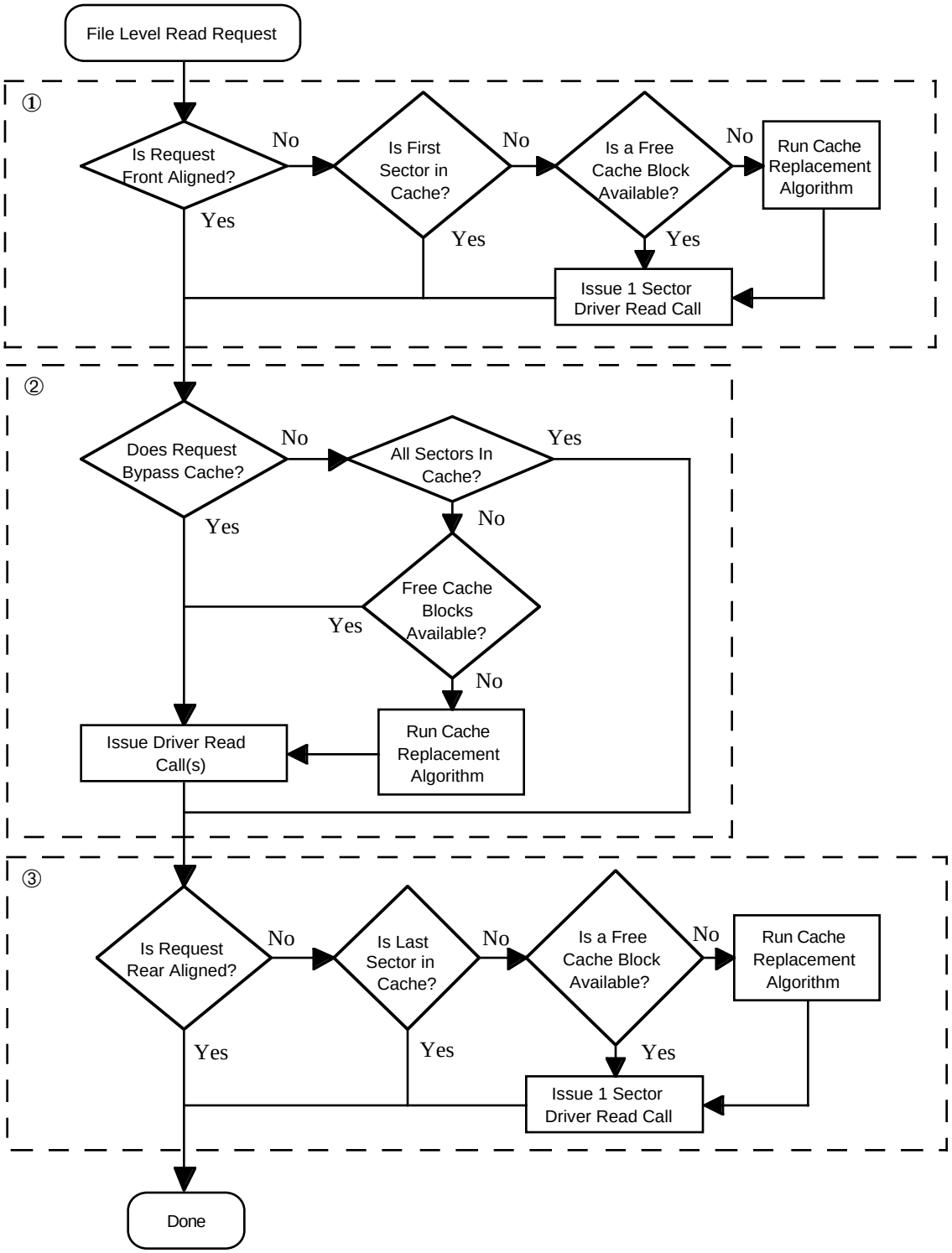

Figure 8. File Level Read Logic

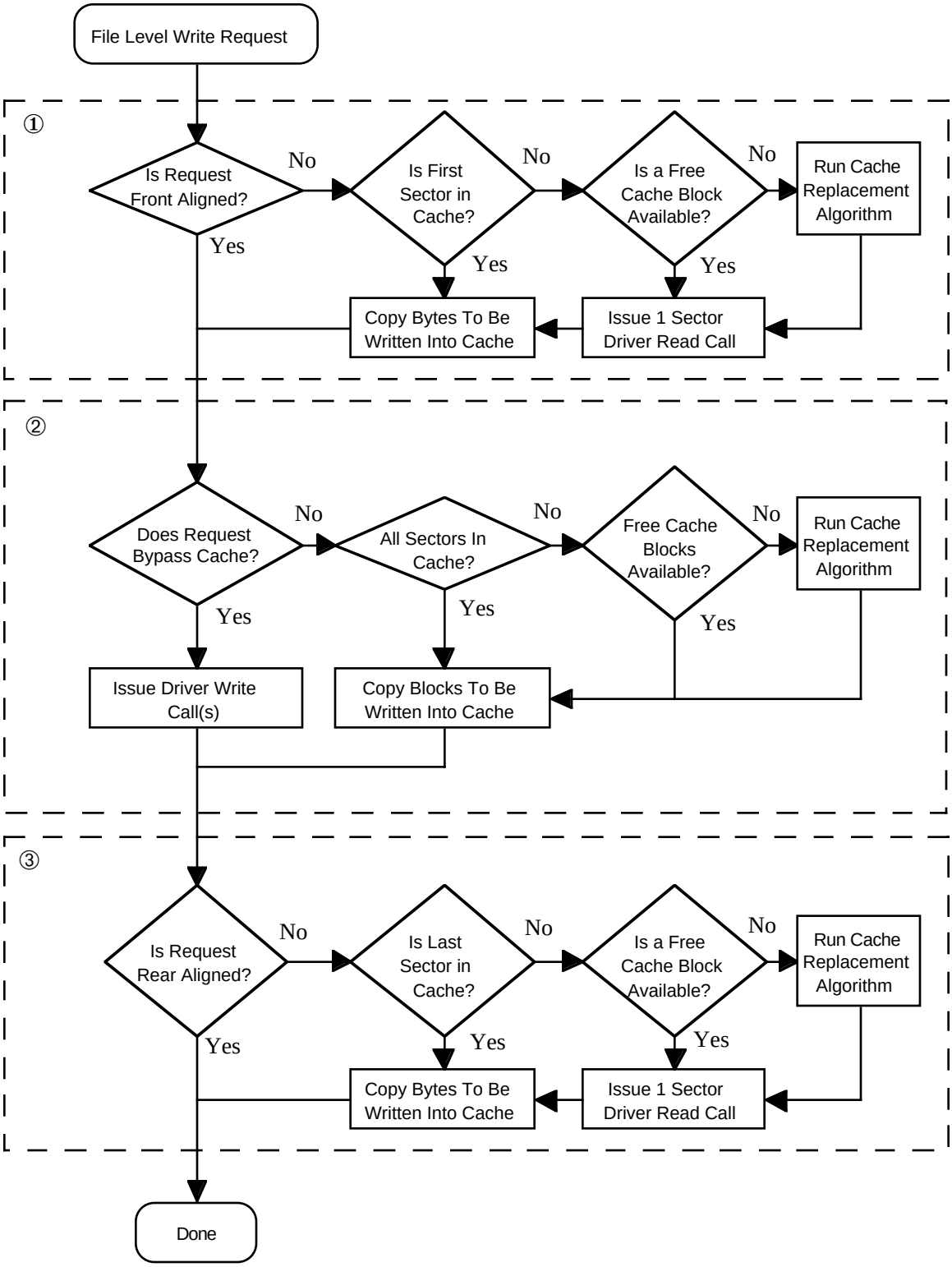

Figure 9. File Level Write Logic