Effective Use of Game Disc File Caching

By Landon Dyer
Principal Software Development Engineer
Xbox Platform Group

Published: January 10, 2007

Many Xbox 360 titles could make more effective use of the hard disk drive than they do. There are several reasons why the hard disk drive is underutilized. For example, not all Xbox 360 consoles have hard disk drives. Therefore titles must be designed to meet load time requirements by using the optical disc only. This makes use of the hard disk drive less important overall. However, the game disc caching feature of the GDK provides a simple and effective use of the utility drive on the hard disk. This feature requires very little development effort. It also improves performance and user experience.

This white paper covers the following topics:

- Problem and Solution
- Technical Overview of Caching
- Quick Start
- Advanced Cache Control
- Guidelines
- Recommendations and Cautions
- Summary

Problem and Solution

The Xbox 360 hard disk drive offers significantly better access time and transfer rate than the game disc. It provides eight to ten times faster seek performance, and up to twice the maximum transfer rate of the DVD drive. Locating data on multiple drives (for example, to access different resources simultaneously) can also improve performance and user experience.

Game disc caching uses the 2 GB utility drive that is reserved for your title. You choose the files to cache, and the library does the rest. Game disc file open calls are transparently redirected to files that have been automatically copied to the hard drive cache. If you don't identify which files to cache, the library attempts to determine them for you.

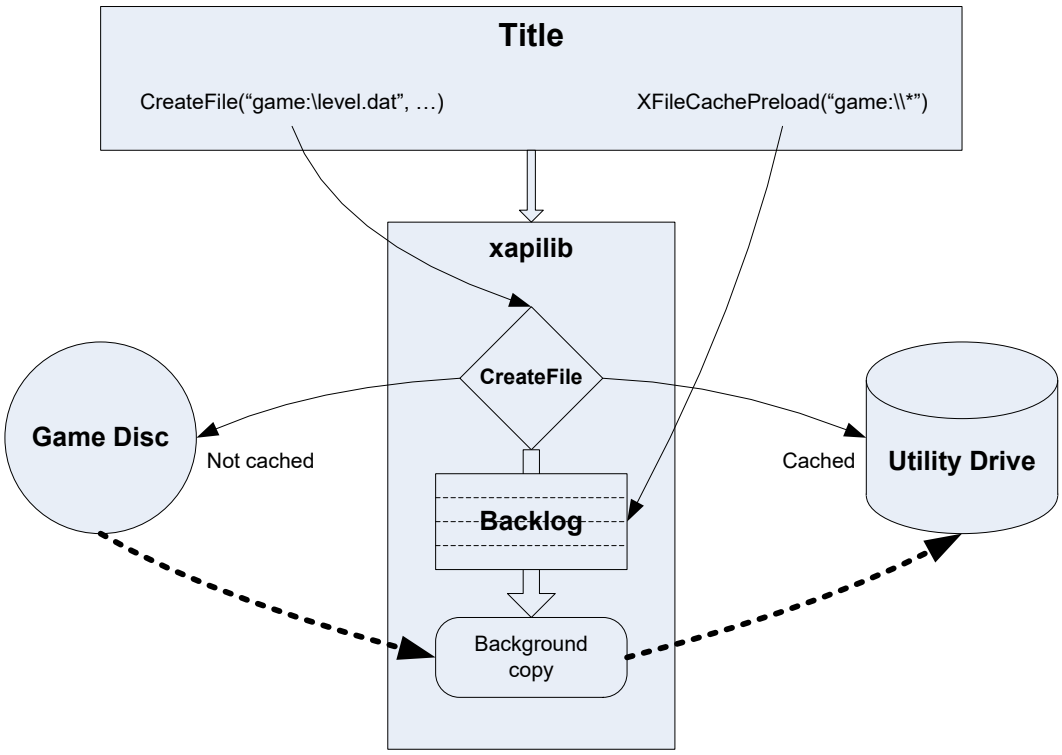
This feature is not a substitute for a custom hard disk caching scheme that incorporates deep knowledge of your title's data structures and access patterns. The library caches at the granularity of whole files. Therefore, it can not decide when to load objects into memory. But if you did not plan to include caching support in your title because of the development cost, then you may find that this library provides noticeable performance improvements for very little investment.

Technical Overview

The cache consists of three components that are statically linked to your title. They are:

- A set of interceptions for file system calls, such as **CreateFile** and **ReadFile**.
- A worker thread (the background copy engine), which is responsible for populating the cache.
- API calls that control the cache.

The following diagram shows the general flow of control after you initialize the cache.



When you open a file by using **CreateFile**, the caching library checks to see whether the file is already in the cache. If it is in the cache, then a handle to the cached file is returned. Otherwise, **CreateFile** notes that you tried to open the specified file, and adds it to a backlog. It then opens the file on the game disc as usual. The background copy engine copies the file to the cache when resources are available.

In addition, you can explicitly add files to the backlog by calling **XFileCachePreload**.

Copying is done in small pieces. The background copy engine identifies times during which your title is not accessing the game disc. It uses the available time to copy the backlog of files to the cache. Your title can completely control the background engine. You can tell it to hold off for a specified period of time, or you can disable and reenable it as necessary.

On consoles that do not have hard disks, the entire caching API fails gracefully, and simply returns success. You can call functions such as **XFileCachePreload** without worrying about whether a hard disk is present.

Quick Start

To get started, you simply need to make one call to an initialization function. All other calls are optional.

```
#include "XFileCache.h"

// At title start

DWORD err = XFileCacheInit(0,     // Flags
                           0,     // MaxCacheSize
                           5,     // HardwareThread
                           0,     // ScratchBufferSize
                           0);    // AppVersion
```

To load specific files into the cache, use **XFileCachePreload**. For example, the following call preloads all files in the subdirectory named myStartup that is located in the title media:

```
err = XFileCachePreload(XFILECACHE_STARTUP_FILES,
                        "game:\\myStartup\\*");
```

You can make this call multiple times, or you can use **XFileCachePreloadFiles,** which lets you specify more than one file.

Shutdown

It is a good idea to shut the cache down before you launch another component of your title.

```
XFileCacheShutdown();
```

Advanced Cache Control

When the game disc cache detects a read from the DVD, the cache stops reading the DVD for at least 250 milliseconds. This helps to prevent DVD access conflicts between your title and the cache.

However, there may be situations where the default backoff doesn't work with your title, or where your title cannot afford to lose any control of the DVD. There are two ways to control these situations. One is to use **XFileCacheControl**. The other is to use **XFileCacheSetFileIoCallbacks**.

You may also find it useful to prevent caching for specific files.

DVD Access Coordination with XFileCacheControl

An explicit call to **XFileCacheControl** can be used to enable or disable the cache's DVD access. The following call turns the background copy engine off:

```
XFileCacheControl(XFILECACHE_BACKGROUND_OFF);
```

The following call turns it back on again:

```
XFileCacheControl(XFILECACHE_BACKGROUND_ON);
```

DVD Access Coordination with XFileCacheSetFileIoCallbacks

Another way to control cache access to the DVD is to call **XFileCacheSetFileIoCallbacks**. The callback is called if the cache needs advice about whether it should read from the DVD. The callback function returns a value that tells the copy engine whether it can read from the DVD. It also specifies the duration of DVD access.

To install a callback, supply a function and a context to **XFileCacheSetFileIoCallbacks**:

```
XFileCacheSetFileIoCallbacks(MyThrottleCallback,
                             NULL);       // Context
```

The response of this function is one of two values—permission or refusal to read the DVD for a specified number of milliseconds:

```
DWORD MyThrottleCallback(
   VOID*           Context,               // your context parameter
   const char*     Path,                  // NULL, or file path
   LARGE_INTEGER*  pFileOffset,           // NULL, or file offset
   DWORD           nNumberOfBytesToRead   // size of request
  )
{
    if ( your title cannot afford any other DVD accesses )
    {
        //  tell the cache to ask again in two seconds
        return XFILECACHE_READFILEADVISORY_HOLDOFF(2000);
    }
   else
   {
        //  give the cache one second of read time
        return XFILECACHE_READFILEADVISORY_ALLOW(1000);
    }
}
```

Note that the parameters **Path** and **pFileOffset** may be NULL. They must be tested. They are intended to provide a hint about the file that is being read, but they are not guaranteed to be present.

Note also that this callback may occur in any thread. You will almost certainly have to use some kind of synchronization to dig into your title's internal data structures.

The callback is called before each file copy is started—at least as often as the advisory time that is returned. The callback may occur earlier than you specify. For example, if you return an instruction to hold off for thirty seconds, and if the copy engine finishes one file and moves on to another, you may get a query callback before the period ends.

Preventing Caching for Specific Files

Normally, all calls to **CreateFile** are candidates for caching. However, there are files that you may not want to cache. For example, movies that are played only once are probably not good candidates for caching.

The function **XFileCacheCreateUncachedFile** has the same parameters as **CreateFile**, with the following exceptions:

- It ensures that the handle that it returns is never to the cached copy.
- The file that it opens is never considered for copying to the cache.

Because a game must be able to run in an environment where no hard drive is available, some files are explicitly run from the DVD. Avoid caching files that must run from the DVD in all scenarios, such as streaming video or streaming audio.

Guidelines

The following guidelines can help you make the best use of the Xbox 360 file caching functions.

File Characteristics

Game Disc Caching works best if your title meets the following criteria:

- It does not contain many small files. Several hundred largish files should work well, while several thousand very small files result in increased memory usage and poor utilization of the hard disk.
- It includes easily identifiable periods of time during which it is possible to access the DVD drive without affecting your title's performance. Dead time, for example during user menu interaction, is ideal for copying. On the other hand, a title that continually streams music does not give the caching system a chance to read from the DVD.

If you want to cache very large files, it may take quite some time for the files to be copied.

Clearing the Cache During Development

If you turn on caching, you may encounter the following minor development issue.

The caching system is designed to cache files that are provided on read-only media. For performance reasons it does not check to see if cached files have changed on the DVD. However, your environment during development may be different. If you have modified a

level on your disc image, for example, you must make sure to clear the cache when your title starts up. To clear the cache, pass **XFILECACHE_CLEAR_ALL** to **XFileCacheInit**.

The cache is predicated on your title's version number, which is found in default.xex. If your title is updated, the version number changes and the cache is automatically cleared. This ensures that cached data remains up to date

Sharing the Utility Drive

You can reserve as much of the available space on the utility drive as you want for your title—up to the 2 GB maximum—and the cache uses the rest. One parameter to **XFileCacheInit** is **MaxCacheSize**, which controls how much of the utility drive is available for caching. If you use more than the amount you specify, the cache simply stops copying until space is free. It then resumes copying.

To use caching and put your own files on the utility drive at the same time, do the following:

1. Call **XFileCacheInit**. Do not call **XMountUtilityDrive**.
2. Call **XFlushUtilityDrive** after creating, writing or deleting files or directories on the utility drive. Do not call **XFlushUtilityDrive** if the cache drive is not present.

Heavy dynamic use of the utility drive—such as creating many small files, or mixing many small files and large files—may result in file system fragmentation over time. One solution is to clear the utility drive occasionally.

Utility Drive Performance

On current development kits, the utility drive is located on the slower, inner tracks of the hard disk. It typically yields reads between 18 and 20 MB per second. (The hard drive is guaranteed to have a minimum sustained data rate of 17 MB per second.) On retail units, the utility drive is located on the fastest part of the disk. Current retail consoles have much higher throughput, which approaches 30 MB per second. Therefore, you may experience some performance degradation when you run your title on the development kit disk layout.

It is important to measure performance on a configuration that closely resembles the retail environment. Here's how:

- Recover your development kit to retail mode. This most closely resembles a retail kernel.
- Use DVD emulation with emulated latency turned on.

The development kit places cached files in a directory on the utility drive that is named as follows:

```
cache:\$cache$\leafname_identifier
```

The **leafName** is part of the leaf name of the cached file. If the leaf name is long, only part of it is used. The **identifier** is a unique number. By using the leaf name fragment and the file size, you can determine the cached file.

Note: The storage scheme for cached files may change with new releases of the GDK. Do not create dependencies on any of the files in this directory.

File Event Monitor shows the following kinds of access to cached files:

- Opens and reads that are made by your title.
- Reads from the DVD and writes to the hard drive that are made by the background copy engine.

File Event Monitor traces can tell you if the copy engine's access affects the access by your title. If you see DVD access by both your title and the copy engine—particularly with large seek deltas—consider adding backoff logic in a callback function that is registered by using **XFileCacheSetFileIoCallbacks**.

Clearing the Cache During Play

The fifth parameter to the **XFileCacheInit** function, **AppVersion**, is a DWORD version number that is completely under your control. This value is stored in the cache when it is initialized. If the value you supply does not match the stored value, the cache is cleared. This provides an easy way for you to ensure that the cache does not contain stale data. For example, if your build system has a build number, simply pass the number.

Whether your title uses the file caching library or another system to provide caching, it is a good idea to provide a way to clear the cache explicitly. Because users seldom shut down Xbox 360 in an orderly fashion, explicit cache clearing is important to clear a badly fragmented file system or corrupt cache structures. One simple approach is to check whether both shoulder buttons are held down when the game starts.

Not for DLLs

The cache is implemented in a static library. This means it works with the following constraints:

- You can utilize disc caching—that is, call **XFileCacheInit**—only from your main executable.
- When caching is enabled, file handles have a different internal type. The handles that are returned from **CreateFile** must not be passed to DLLs, even if the other DLLs have been statically linked with the caching library.
- It is okay to use file handles that are returned to you by DLLs, because they are always uncached. You can also use handles that are the result of calling **XFileCacheCreateUncachedFile** in any way you want.

These constraints do not preclude breaking up your title into separate executables. For example, you can create a shell executable that launches the real title. This executable in turn launches the shell when it quits.

Either or both of these executables can use the caching library.

Remember to call **XFileCacheShutdown** before you launch the next component.

Resource Use

The caching feature uses XMemAlloc to obtain an initial 192 KB or so of memory for buffers. An additional 64 KB or so of smaller allocations are taken dynamically by **XMemAlloc**. If your title has many files, you will see allocations beyond these. There is a small amount of overhead for each open file.

The copy engine spends most of its time waiting for I/Os to complete. You should not see it do much computation.

Recommendations and Cautions

The following lists are intended to help you maximize your use of the game disc caching feature.

Do

- Provide preloading hints when you use **XFileCachePreload**. This helps to focus the cache on files that you know are important.
- Clear the cache during development. Stale files that are held in the cache are not invalidated by newer files on the media.
- Analyze your DVD access patterns by using the File Event Monitor pane in PIX. Use the results to determine whether you need to use **XFileCacheSetFileIoCallbacks** to better control the cache.
- Use appropriate locking in your cache control callback.
- Call **XFileCacheShutdown** before you call **XLaunchNewImage** to launch another component of your title.

Don't

- Put thousands of files on a DVD. Hundreds of large files are okay. Many thousands of smaller files make file opens more expensive.
- Use game disc caching from a DLL. Multiple instances of game disc caching cannot coexist, because the feature is implemented by using a statically linked library.
- Share handles opened when caching is turned on.
- Call **XFlushUtilityDrive** unless you know that the utility drive is present. If there is no utility drive, calling this results in an assert in the debug versions of the libraries. However, in the release versions it simply returns an error.

Summary

The dedicated 2 GB utility drive on the Xbox 360 development console allows you to take advantage of the game disc caching feature. If you had not planned to use the hard disk,

this feature is a good way to improve load times without having to write a lot of your own code. You can make simple calls to pre-populate the cache with your files. You can also control the behavior of the cache to meet the exact needs of your title.