Advanced Authoring Format

TN09 – I/O Performance and the AAF Toolkit.

April 5, 2004 Tim Bingham

Abstract

This Technical Note describes how to get the best I/O performance from the AAF Toolkit. It describes the performance improvement options available to you as an AAF Toolkit client programmer.

Audience

All AAF Developers

Revision History

Revision	Description	Author/ Contributors	Date
1.0	Initial revision	Tim Bingham	7/16/2003
1.1	Add information on caching and file open and create latency.	Tim Bingham	7/17/2003
1.2	Make it clear that IAAF{Random}RawStorage is used for both metadata and essence.	Tim Bingham	9/5/03

1.3	Add note on how to determine quickly if a given file is an AAF file.	Tim Bingham	2/24/04
1.4	Explicitly state that the ISO C I/O functions don't support large files (typically the limit is 2Gb).	Tim Bingham	4/5/04

Performing your own I/O

To get the best performance from the toolkit you should perform the I/O yourself avoiding using the built in I/O routines. Typically you will have spent time developing and optimizing your application's I/O routines to meet your performance constraints and to support large files. Using the techniques described in this note you can have the AAF toolkit call those routines when you read or write essence or metadata objects via the toolkit. Additionally you control the file system calls to open/create and close the file at the file system level. Note that standard toolkit interfaces are used and that no modifications to the toolkit are required.

What are the built in I/O routines

There are two sets of built in I/O routines -

- those built in to the toolkit and coded to use the ISO C I/O library, and
- those built in to Microsoft Structured Storage

You get the built in I/O routines if you create/open the file with -

- AAFFileOpenExistingRead
- AAFFileOpenExistingModify
- AAFFileOpenNewModify
- AAFFileOpenNewModifyEx

You also get the built in I/O routines if you create/open the file with AAFCreateAAFFileOnRawStorage() using one of the built-in raw storage implementations created with -

- AAFCreateRawStorageDisk
- AAFCreateRawStorageCachedDisk

What's wrong with the built in I/O routines

The built in Microsoft Structured Storage I/O routines as very fast on a local Windows disk but have the following drawbacks –

- The Macintosh implementations are slow
- The Windows implementations are optimized for local Windows disks and perform poorly on remote volumes and/or non-Windows file systems such as Unity

The built in toolkit routines are

- coded for portability using the ISO C I/O library, the routines in this library
    - are typically slow
    - don't support large files. Typically the size limitation is 2Gb (sizeof(long int) == 4)

How the AAF Toolkit accesses your I/O routines

The AAF toolkit calls your I/O routines through the IAAFRawStorage and IAAFRandomRawStorage interfaces which you implement using your routines. When you create or open an AAF file using the toolkit you pass in your implementation of these interfaces. When the toolkit needs to perform I/O it does so through these interfaces and your I/O routines are called.

Recommended Approach

The recommended approach is as follows -

- Create/open the AAF file using AAFCreateAAFFileOnRawStorage()
- Pass to AAFCreateAAFFileOnRawStorage() a custom implementation of IAAF{Random}RawStorage, this allows you to use your own I/O routines.

- If caching is desired then it can be added to your custom implementation of IAAF{Random}RawStorage by calling AAFCreateRawStorageCached()

The sequence of calls is thus –

- Open/create a file in the file system
- Create an instance of your custom raw storage from the open file
- AAFCreateRawStorageCached()
- AAFCreateAAFFileOnRawStorage()
- IAAFFile::Open()
- Add objects to the AAF file
- IAAFFile::Save()
- IAAFFile::Close()
- Close the file in the file system

What Functions do you need to implement ?

The IAAFRawStorage interface defines the following routines -

- IsReadable
- Read
- IsWriteable
- Write
- Synchronize

The IAAFRandomRawStorage interface defines the following routines –

- ReadAt
- WriteAt
- GetSize
- IsExtendable
- GetExtent
- SetExtent

In addition you will need to –

- open/create a file system object (e.g. file handle)
- construct an instance of your raw storage implementation from the file system object
- close the file system object

See the source file ref-impl/include/ref-api/AAF.h for details.

Caching

You may choose to apply caching to your implementation of IAAF{RandomRawStorage using the function AAFCreateRawStorageCached . This function will apply a page cache to your raw storage. You can specify the page size and the number of pages to cache. Choosing a page size that is the same as or a multiple of the disk sector size is usually a good idea. The cache is a page cache with LRU replacement.

If you do choose to apply a cache, your raw storage implementation will receive read and write calls for whole pages only (except for the last page in the file).

To use caching effectively you should be aware of the following points –

- There's currently no way for your raw storage implementation and/or the cache to distinguish references to essence from references to metadata. While caching provides improvements for metadata you will probably not want to use caching for essence. Therefore use of the page cache is best reserved for AAF files containing only metadata (i.e. with linked essence).

- If the operating system and or your raw storage implementation is already using caching the addition of caching within the AAF toolkit by using AAFCreateRawStorageCached, might actually make performance worse.

Determining quickly if a file is an AAF file

The most general way to determine if a given file is an AAF file is to call AAFFileIsAAFFile. This function returns true (through the pFileIsAAFFile output parameter) if the file is an AAF file, and, if so, the encoding (or file kind) through the pAAFFileKind output parameter.

If you are only interested in, say, AAF files encoded as structured storage it is faster to call AAFFileIsAAFFileKind, which returns true (through the pFileIsAAFFile out put parameter) if the file is an AAF file encoded as specified by the pAAFFileKind *input* parameter.

The function AAFFileIsAAFFile checks the file against all known file kinds while the function AAFFileIsAAFFileKind checks only against the specified file kind.

The performance difference between AAFFileIsAAFFile and AAFFileIsAAFFileKind is significant when scanning large numbers of files.

Each function has a corresponding function that operates on an IAAFRawStorage instead of a named file. For AAFFileIsAAFFile the corresponding function is AAFRawStorageIsAAFFile, and for AAFFileIsAAFFileKind the corresponding functions is AAFRawStorageIsAAFFileKind.

Note that these functions check only that a file purports to be an AAF file (usually by checking a signature at the beginning); they don't perform any validation of the file contents.

Example Code

See the source file test/com/ComModTestAAF/ModuleTests/CAAFRandomRawStorageTest.cpp