JFS Tuning and Performance

Mark Ray
Hewlett Packard Company

© 2004 Hewlett-Packard Development Company, L.P.
The information contained herein is subject to change without notice

August 2004

Introduction

Beginning with HP-UX 10.0, Hewlett-Packard began providing a Journaled File System (JFS) from VERITAS Software Corporation known as the VERITAS® File System™ (VxFS®)[1]. You no longer need to be concerned with performance attributes such as cylinders, tracks, and rotational delays. The speed of storage devices have increased dramatically and computer systems continue to have more memory available to them. Applications are becoming more complex, accessing large amounts of data in many different ways.

In order to maximize performance with JFS file systems, you need to understand the various attributes of the file system, and which attributes can be changed to best suit how the application accesses the data.

This paper is based on the HP JFS 3.3[2] product, currently available on HP-UX 11.0 and above, and has been updated to include changes for HP JFS 3.5.

Many factors need to be considered when tuning your JFS file systems for optimal performance. The answer to the question "How should I tune my file system for maximum performance?" is "It depends". Understanding some of the key features of JFS and knowing how the applications access data are keys to deciding how to best tune the file systems.

The following topics are covered in this paper:

- Understanding JFS
- Understanding your application
- Creating your file system
- Mount options
- File system tunables
- System wide tunables
- JFS ioctl() options

Understanding JFS

In order to fully understand some of the various file system creation options, mount options, and tunables, a brief overview of JFS is provided.

---

1. VERITAS is a registered trademark of VERITAS Software Corporation

2. JFS will be used to refer to the HP products HP OnlineJFS/JFS, while VxFS will be used to refer to the filesystem type used by the Veritas Filesystem.

Software Versions vs. Disk Layout Versions

JFS supports several different disk layout versions. The default disk layout version can be overridden when the file system is created using mkfs(1M). Also, the disk layout can be upgraded online using the vxupgrade(1M) command.

OS	SW version	Disk layout version
10.01	JFS 2.0	2*
10.10	JFS 2.3	2*
10.20	JFS 3.0	2,3*
11.0	JFS 3.1 JFS 3.3	2,3* 2,3*,4
11.11	JFS 3.3 JFS 3.5	2,3,4*
11.22	JFS 3.3	2,3,4*
11.23	JFS 3.5	2,3,4,5*

Table 1: JFS software versions and disk layout versions
***denotes default disk layout**

Several improvements have been made in the disk layout which can help increase performance. For example, files that extend into indirect extents have a new format with the version 3 disk layout that makes handling large files much more efficient as well as reduces fragmentation. The version 4 disk layout allows for extent reorganization of large directories, potentially reorganizing the large directory into large direct extents and eliminating indirect extents. The reorganization could reduce contention on the indirect blocks when multiple threads are performing directory lookups simultaneously.

Please note that you cannot port a file system to a previous release unless the previous release supports the same disk layout version for the file system being ported. Before porting a file system from one operating system to another, be sure the file system is properly unmounted. Differences in the Intent Log will likely cause a replay of the log to fail and a full fsck will be required before mounting the file system.

Variable Sized Extent Based Filesystem

JFS is a variable sized extent based file system. Each file is made up of one or more extents that vary in size. Each extent is made up of one or more file system blocks. A file system block is

the smallest allocation unit of a file. The block size of the file system is determined when the file system is created and cannot be changed without rebuilding the filesystem. The default block size varies depending on the size of the file system.

The advantages of variable sized extents include:

- Faster allocation of extents
- Larger and fewer extents to manage
- Ability to issue large physical I/O for an extent

However, variable sized extents also have disadvantages:

- Free space fragmentation
- Files with many small extents

Extent Allocation

When a file is initially opened for writing, JFS is unaware of how much data the application will write before the file is closed. The application may write 1k of data or 500 MB of data. The size of the initial extent is the largest power of 2 greater than the size of the write, with a minimum extent size of 8k. Fragmentation will limit the extent size as well.

If the current extent fills up, the extent will be extended if there is neighboring free space. Otherwise, a new extent will be allocated that is progressively larger as long as there is available contiguous free space.

When the file is closed by the last process that had the file opened, the last extent is trimmed to the minimum amount needed.

Fragmentation

Two types of fragmentation can occur with JFS. The available free space can be fragmented and individual files may be fragmented. As files are created and removed, free space can become fragmented due to the variable sized extent nature of the file system. For volatile file systems with a small block size, the file system performance can be degraded significantly if the fragmentation is severe. Examples of applications that use many small files include mail servers. As free space becomes fragmented, file allocation takes longer as smaller extents are allocated. Then, more and smaller I/Os are necessary when the file is read or updated.

However, files can be fragmented even when there are large extents available in the free space map. This fragmentation occurs when a file is repeatedly opened, extended, and closed. When the file is closed, the last extent is trimmed to the minimum size needed. Later, when the file is extended again but the last extent cannot be extended, a new extent will be allocated and the process will repeat. This can result in a large file with many small extents.

Static file systems that use large files built shortly after the file system is created are less prone to fragmentation, especially if the file system block size is 8k. Examples are file systems that have large database files. The large database files are often updated but rarely change in size.

File system fragmentation can be reported using the "-E" option of the fsadm(1M) utility. File systems can be defragmented or reorganized using the HP OnLineJFS product using the "-e" option of the fsadm utility. Remember, performing 1 8k I/O will be faster than performing 8 1k I/Os. File systems with small block sizes are more susceptible to performance degradation due to fragmentation than file systems with large block sizes.

File system reorganization should be done on a regular and periodic basis, where the interval between reorganizations depends on the volatility, size and amount of the files. Large file systems with a large number of files and significant fragmentation can take an extended amount of time to defragment. The extent reorganization is run online, however, increased disk I/O will occur when fsadm copies data from smaller extents to larger ones.

The file system reorganization attempts to collect large areas of free space by moving various file extents and attempts to defragment individual files by copying the data in the small extents to larger extents. The reorganization is not a compaction utility and does not try to move all the data to the front on the file system.

Be aware that the fsadm extent reorganization needs some large free areas in order to be effective. A file system could be too fragmented to be reorganized.

Transaction Journaling

By definition, a journal file system logs file system changes to a "journal" in order to provide file system integrity in the event of a system crash. JFS logs structural changes to the file system in a circular transaction log called the ***Intent Log***. A ***transaction*** is a group of related changes that represent a single operation. For example, adding a new file in a directory would require multiple changes to the file system structures such as adding the directory entry, writing the inode information, updating the inode bitmap, etc. Once the Intent Log has been updated with the pending transaction, JFS can begin committing the changes to disk. When all the disk changes have been made on disk, a completion record is written to indicate that transaction is complete.

With the *datainlog* mount option, small synchronous writes are also logged in the Intent Log. The *datainlog* mount option will be discussed in more detail later in this paper.

Since the Intent Log is a circular log, it is possible for it to "fill" up. This occurs when the oldest transaction in the log has not been completed, and the space is needed for a new transaction. When the Intent Log is full, the threads must wait for the oldest transaction to be flushed and the space returned to the Intent Log. A full Intent Log occurs very infrequently, but may occur more often if many threads are making structural changes or small synchronous writes (*datainlog*) to the file system simultaneously.

Understanding Your Application

As you plan the various attributes of your file system, you must also know your application and how your application will be accessing the data. Ask yourself the following questions:

- Will the application be doing mostly file reads, file writes, or both?
- Will it be doing sequential I/O or random I/O?
- What size will the I/Os be?
- Does the application use many small files, or a few large files?
- How is the data organized on the volume or disk? Is the file system fragmented? Is the volume striped?
- Are the files written or read by multiple processes or threads simultaneously?
- Which is more important, data integrity or performance?

How you tune your system and file systems depend on how you answer the above questions. There is no single set of tunables that can be applied to all systems such that all systems will run at peak performance.

Creating Your Filesystem

When you create a file system using newfs(1m) or mkfs(1m), you need to be aware of several options that could affect performance.

Block Size

The block size (***bsize***) is the smallest amount of space that can be allocated to a file extent. Most applications will perform better with an 8k block size. Extent allocations are easier and file systems with an 8k block size are less likely to be impacted by fragmentation, since each extent would have a minimum size of 8k. However, using an 8k block size could waste space, as a file with 1 byte of data would take up 8k of disk space. The default block size for a file system scales depending on the size of the file system.

The following table documents the default block sizes for the various JFS versions:

FS Size	JFS 3.1	JFS 3.3	11.11 JFS 3.5	11.23 JFS 3.5
< 8GB	1K	1K	1K	1K
<16GB	1K	2K	1K	1K
<32GB	1K	4K	1K	1K
<=2TB	1K	8K	1K	1K
<=4TB	-	-	-	1K[3]
<=8TB	-	-	-	2K[3]
<=16 TB	-	-	-	4K[3]
<=32TB	-	-	-	8K[3]

Table 2: Default FS block sizes

Also in the JFS 3.5 case, the default block size is also the minimum block size that can be used the file system. For example, if a 32TB file system is created, it MUST use a block size of 8kb.

For efficient Direct I/O processing, each I/O should be aligned on a file system block boundary. Note in some applications where a 1Kb block size is recommended, a 1Kb block size is impossible due to the FS size.

Note that outside of Direct I/O, the block size is not important as far as reading and writing data is concerned. The block size is an allocation policy parameter, not a data access parameter.

Intent Log Size

The default size of the Intent Log (***logsize***) scales depending on the size of the file system. All file systems >8 MB will have an Intent Log size of 1MB. For most applications, the default log size is sufficient. However, large file systems with heavy structural changes simultaneously by multiple threads or heavy synchronous write operations with ***datainlog*** may need a larger Intent Log to prevent transaction stalls when the log is full.

Disk Layout Version

As mentioned earlier, there have been several improvements in the disk layout which can help improve performance. Examples include a new inode format for files extending into indirect extents (version 3) and extent reorganization of large directories (version 4).

---

[3] VxFS disk version layout 5 (11.23) and HP OnlineJFS license is needed to create filesystems past 2TB. Currently, HP supports filesystems <= 12 TB.

See the manpages mkfs_vxfs(1M) and newfs_vxfs(1M) for more information on file system create options.

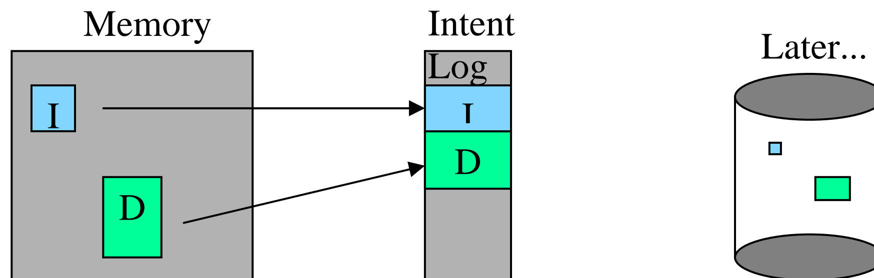
Mount Options

blkclear

When new extents are allocated to a file, extents will contain whatever was last written on disk until new data overwrites the old uninitialized data. Accessing the uninitialized data could cause a security problem as sensitive data may remain in the extent. The newly allocated extents could be initialized to zeros if the ***blkclear*** option is used. In this case, performance is traded for data security.

When writing to a "hole" in a sparse file, the newly allocated extent must be cleared first, before writing the data. The ***blkclear*** option does not need to be used to clear the newly allocated extents in a sparse file. This clearing is done automatically to prevent stale data from showing up in a sparse file.

datainlog, nodatainlog



When using HP OnLineJFS, small (<= 8k) synchronous writes are logged to the Intent Log using the mount option ***datainlog***. Logging small synchronous writes provides a performance improvement as several transactions can be flushed with a single I/O. Once the transaction is flushed to disk, the actual write can be performed asynchronously, thus simulating the synchronous write. In the event of a system crash, the replay of the intent log will complete the transaction and make sure the write is completed. However, using ***datainlog*** can cause a problem if the asynchronous write fails to complete, possibly due to a disk I/O problem. If the I/O is returned in error, the file will be marked bad and cleaned up (removed) during the next run of fsck. While ***datainlog*** provides a performance improvement for synchronous writes, if true synchronous writes are needed, then the ***datainlog*** option should ***not*** be used.

Using ***datainlog*** has no affect on normal asynchronous writes. The option ***nodatainlog*** is the default for systems without HP OnlineJFS, while ***datainlog*** is the default for systems that do have HP OnlineJFS.

mincache options

The ***mincache*** options control how standard asynchronous I/O is performed.

The mincache option ***closesync*** flushes a file's buffers when the file is closed. The close() system call may experience some delays while the buffers are flushed, but the integrity of closed files would not be compromised by a system crash.

The mincache option ***dsync*** converts asynchronous I/O to synchronous I/O. For best performance the mincache=dsync option should not be used. This option should be used if the data must be on disk when write() system call returns.

The mincache options ***direct*** and ***unbuffered*** are similar to the ***mincache=dsync*** option, but all I/Os are converted to direct I/O. For best performance, mincache=direct should not be used unless I/O is large and expected to be accessed only once, or synchronous I/O is desired. All direct I/O is synchronous and read ahead and flush behind are not performed.

The mincache option ***tmpcache*** provides the best performance, but less data integrity in the event of a system crash. The ***mincache=tmpcache*** option does not flush a file's buffers when it is closed, so a risk exists of losing data or getting garbage data in a file if the data is not flushed to disk before a system crash occurs.

All the mincache options except for closesync require the HP OnlineJFS product.

convosync options

The ***convosync*** options convert synchronous (O_SYNC) operations so they behave differently. Normal asynchronous operations are not affected. The convosync options are only available with the HP OnLineJFS product.

The convosync option ***closesync*** converts O_SYNC operations to be asynchronous operations. The file's dirty buffers are then flushed when the file is closed. While this option speeds up applications that use O_SYNC, it may be harmful for applications that rely on data to be written to disk before the write() system call completes.

The convosync option ***delay*** converts all O_SYNC operations to be asynchronous. This option is similar to convosync=closesync except that the file's buffers are not flushed when the file is closed. While this option will improve performance, applications that rely on the O_SYNC behavior may fail after a system crash.

The convosync option ***dsync*** converts O_SYNC operations to O_DSYNC operations. Data writes will continue to be synchronous, but the associated inode time updates will be performed asynchronously.

The convosync options ***direct*** and ***unbuffered*** cause O_SYNC operations to bypass buffer cache and perform Direct I/O. These options are similar to the ***convosync=dsync*** option as the inode time update will be delayed.

Intent Log options

There are 4 levels of transaction logging:

- ***nolog*** – Originally, no transaction logging was performed. With JFS 3.3, nolog is equivalent to tmplog.
- ***tmplog*** – Most transaction flushes to the Intent Log are delayed, so some recent changes to the file system will be lost in the event of a system crash.
- ***delaylog*** – Some transaction flushes are delayed.
- ***log*** – Most all structural changes are logged before the system call returns to the application.

Tmplog, delaylog, log options all guarantee the structural integrity of the file system in the event of a system crash by replaying the Intent Log during fsck. However, depending on the level of logging, some recent file system changes may be lost in the event of a system crash.

There are some common misunderstandings regarding the log levels. For read() and write() system calls, the log level have no affect. For asynchronous write() system calls, the log flush is always delayed until after the system call is complete. The log flush will be performed when the actual user data is written to disk. For synchronous write() systems calls, the log flush always performed prior to the completion of the write() system call. Outside of changing the file’s access timestamp in the inode, a read() system call makes no structural changes, thus does not log any information.

The following table identifies which operations cause the Intent Log to be written to disk synchronously (flushed), or if the flushing of the Intent Log is delayed until some time after the system call is complete (delayed).

Operation	JFS 3.3 JFS 3.5 log	JFS 3.3 delaylog	 JFS 3.5 delaylog	JFS 3.3 JFS 3.5 tmplog
Async Write	Delayed	Delayed	Delayed	Delayed
Sync Write	Flushed	Flushed	Flushed	Flushed
Read	n/a	n/a	n/a	n/a
Sync (fsync())	Flushed	Flushed	Flushed	Flushed
File Creation	Flushed	Delayed	Delayed	Delayed
File Removal	Flushed	Flushed	Delayed	Delayed
File Timestamp changes	Flushed	Delayed	Delayed	Delayed
Directory Creation	Flushed	Flushed	Delayed	Delayed
Directory Removal	Flushed	Flushed	Delayed	Delayed
Symbolic/Hard Link Creation	Flushed	Delayed	Delayed	Delayed
File/Dir Renaming	Flushed	Flushed	Flushed	Delayed

Table 3: Intent Log flush behavior

For example, using JFS 3.3, transactions that are delayed with the delaylog or tmplog options include file creation, creation of hard links or symbolic links, and inode time changes (for example, using touch(1M) or utime() system call). Note that with JFS 3.5, directory creation and removal and file removal are also delayed with delaylog, but on JFS 3.3, the transactions are only delayed with the tmplog option.

Note that using the log mount option has little to no impact on file system performance unless there are large amounts of file create/delete/rename operations. For example, if you are removing a directory with thousands of files in it, it will likely run faster using the delaylog mount option than the log mount option when using JFS 3.5, since the flushing of the intent log is delayed after each file removal. However, with JFS 3.3, the intent log is flushed after each file removal when delaylog is used.

Also, using the delaylog mount option has little or no impact on data integrity, since the log level does not affect the read() or write() system calls. If data integrity is desired, synchronous writes should be performed as they are guaranteed to survive a system crash regardless of the log level.

The ***logiosize*** can be used to increase throughput of the transaction log flush by flushing up to 4Kb at a time. This option is new with JFS 3.5.

The ***tranflush*** option causes all metadata updates (such as inodes, bitmaps, etc) to be flushed before returning from a system call. Using this option will negatively affect performance as all metadata updates will be done synchronously. This option is new with JFS 3.5.

Direct I/O

If OnlineJFS in installed and licensed, direct I/O can be enabled in several ways:

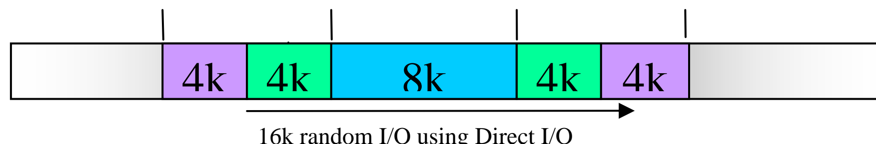
- Mount option "-o mincache=direct"
- Mount option "-o convosync=direct"
- Use VX_DIRECT cache advisory of the VX_SETCACHE ioctl
- Discovered Direct I/O

The advantage of using direct I/O is that data accessed only once does not benefit from being cached. Also, data integrity is enhanced since the disk I/O must complete before the read or write system call can continue.

Direct I/O is typically slower than buffered I/O as all direct I/O is synchronous. When direct I/O is used, no read ahead or flush behind is performed. Each I/O must complete during the system call that initiated the I/O. Direct I/O works best for large I/Os that do not need to be maintained in the buffer cache and the speed of the write is not important.

Direct I/O and Unaligned Data

When performing Direct I/O, alignment is very important. All direct I/O must be aligned on a file system block boundary. Unaligned data is still buffered using 8Kb buffers (default size), although the buffer is not maintained in the buffer cache after the I/O is complete and I/O is still done synchronously.



For large transfers (greater than the block size), parts of the data can still be transferred using direct I/O. For example, consider a 16Kb transfer on a file system using an 8Kb block size. However, the data starts on a 4Kb boundary. Since the 1st 4Kb is unaligned, the data must be buffered. Thus an 8Kb buffer is allocated, and the entire 8Kb of data is read in, thus 4Kb of unnecessary data is read in. Then the next 8Kb is transferred using Direct I/O. Finally, the last 4Kb of data is read into an 8Kb buffer similar to the 1st 4Kb. The result is 3 8Kb I/Os. If the file system was re-created to use a 4Kb block size, the I/O would be aligned and could be performed in a single 16Kb I/O.

Using a smaller block size, such as 1Kb, will improve chances of doing more optimal Direct I/O.

Dynamic File System Tunables

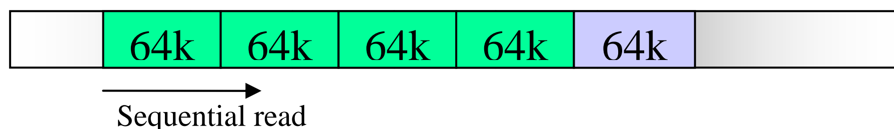
File system performance can be impacted by a number of dynamic file system tunables. These values can be changed online using the vxtunefs(1M) command, or they can be set when the file system is mounted by placing the values in the /etc/vx/tunefstab file (see tunefstab(4)).

Read Ahead

When reads are performed sequentially, JFS detects the pattern and reads ahead or prefetches data into the buffer cache. The size of the read ahead is calculated using two file system tunables as follows:

read ahead size = ***read_pref_io*** * ***read_nstream***

JFS attempts to maintain 4 ranges of data in the buffer cache, where the read ahead size is the size of each range.



Sequential read

The ranges act as a pipeline. When the data in the first of the 4 ranges is read, a read for a new range is initiated.

By reading the data ahead, the disk latency can be reduced. The ideal configuration is the minimum amount of read ahead that will reduce the disk latency. If the read ahead size is configured too large, the disk I/O queue may spike when the reads are initiated, causing delays for other potentially more critical I/Os. Also, the data read into the cache may no longer be in the cache when the data is needed, causing the data to be re-read into the cache. These cache misses will cause the read ahead size to be reduced automatically.

Note that if another thread is reading the file randomly or sequentially, JFS has trouble with the sequential pattern detection. The sequential reads may be treated as random reads and no read ahead is performed. This problem can be resolved with Fancy Read Ahead discussed later. This type of read ahead is the default policy. With JFS 3.5, the read ahead policy can be set on a per-filesystem basis by setting the read_ahead tunable using vxtunefs. The default ***read_ahead*** value is 1 (normal read ahead).

Read Ahead with VxVM Stripes

The default value for read_pref_io is 64Kb, and the default value for read_nstream is 1, except when the file system is mounted on a VxVM striped volume. Then the tunables are defaulted to match the striping attributes. For most applications, the 64kb read ahead size is good (remember, there are 4 ranges of read ahead in progress at any time). Depending on the stripe size (column

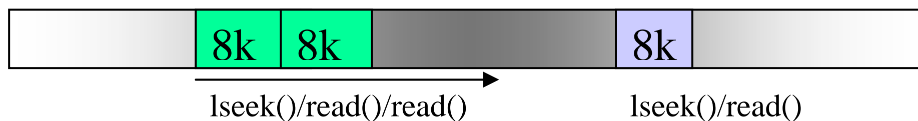
width) and number of stripes (columns), the default tunables on a VxVM volume may cause performance problems.

Consider a VxVM volume with a stripe size of 1MB and 20 stripes. By default, read_pref_io is set to 1MB and read_nstream is set to 20. Thus the calculated read ahead size is 20MB. When JFS initiates the read ahead, it tries to prefetch 4 ranges of the read ahead size, or 80MB. This large amount of read ahead can flood the SCSI I/O queues or internal disk array queues, causing severe performance degradation.

If you are using VxVM striped volumes, be sure to review the ***read_pref_io*** and ***read_nstream*** values to be sure excessive read ahead is not being done.

False sequential I/O patterns and Read Ahead

Some applications trigger read ahead when it is not needed. This occurs when the application is doing random I/O by positioning the file pointer using the lseek() system call, then performing the read() system call.

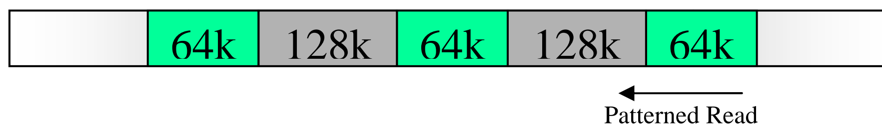


However, if the application issues 2 read() system calls in a row, it triggers the read ahead algorithm in JFS. Although only 2 reads are needed, JFS begins prefetching data from the file, up to 4 ranges of the read ahead size. The larger the read ahead size, the more data read into the buffer cache.

Not only is excessive I/O performed, but useful data that may have been in the buffer cache is invalid so the new unwanted data can be stored.

Fancy Read Ahead

JFS 3.3 introduced a new feature called Fancy Read Ahead. Fancy Read Ahead can detect non-sequential patterns, such as reading every third record or reading a file backwards. Fancy Read Ahead can also handle patterns from multiple threads. For example, two threads can be reading from the same file sequentially and both threads can benefit from the configured read ahead size.



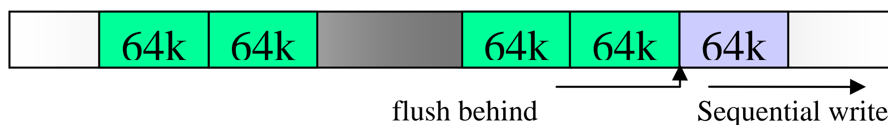
With JFS 3.3, Fancy Read Ahead is enabled by setting the system-wide tunable ***vx_fancyra_enable***. By default, Fancy Read Ahead is enabled on HP-UX 11.0, and patch PHKL_22414 is needed to change the default value. Fancy Read Ahead is disabled by default on HP-UX 11.11 and the default value can be changed without installing a patch.

With JFS 3.5, Fancy Read Ahead can be set on a per-filesystem basis by setting the ***read_ahead*** tunable to 2 with vxtunefs.

Flush Behind

During normal asynchronous write operations, the data is written to buffers in the buffer cache. The buffer is marked "dirty" and control returns to the application. Later, the dirty buffer must be flushed from the buffer cache to disk.

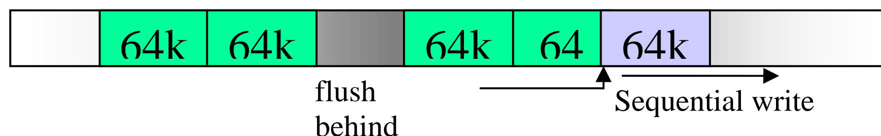
There are 2 ways for dirty buffers to be flushed to disk. One way is through a "sync", either by a sync() or fsync() system call, or by the syncer daemon. The second method is known as "flush behind" and is initiated by the application performing the writes.



As data is written to a JFS file, JFS will perform "flush behind" on the file. In other words, it will issue asynchronous I/O to flush the buffer from the buffer cache to disk. The flush behind amount is calculated by multiplying the ***write_pref_io*** by the ***write_nstream*** file system tunables. The default flush behind amount is 64k.

I/O Throttling

Often, applications may write to the buffer cache faster than JFS and the I/O subsystem can flush the buffers. Flushing too many buffers can cause huge disk I/O queues, which could slow down other critical I/O to the devices. To prevent too many buffers from being flushed simultaneously for a single file, 2 types of I/O throttling are provided.



Flush Throttling (***max_diskq***) (introduced with JFS 3.3)

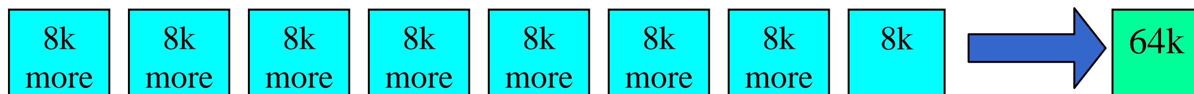
The amount of data (dirty buffers) being flushed per file cannot exceed the ***max_diskq*** tunable. The process performing a write() system call will skip the "flush behind" if the amount of outstanding I/O exceeds the ***max_diskq***. However, when a sync()/fsync() system call is made, the system call cannot return until I/O is initiated to flush all the necessary buffers. Thus during a sync()/fsync() system call, if the amount of outstanding I/O exceeds the ***max_diskq***, the process will wait until the amount of outstanding I/O drops below the ***max_diskq***. Flush throttling can thus cause delays in processes that perform sync operations. The ***max_diskq*** tunable has a default value of 1MB.

Write Throttling (***write_throttle***) (introduced with JFS 3.5)

The amount of dirty data (unflushed) per file cannot exceed ***write_throttle***. If a process tries to perform a write() operation and the amount of dirty data exceeds write_throttle, then the process will wait until some of the dirty data has been flushed. The default value for ***write_throttle*** is 0 (no throttling).

Buffer Sizes

JFS uses a maximum buffer size configured with the file system tunable ***max_buf_data_size*** (default 8k). The only other value possible is 64k. For reads and writes larger than ***max_buf_data_size*** and when performing read ahead, JFS will chain the buffers together when sending them to the Volume Management subsystem. The Volume Management subsystem holds the buffers until JFS sends the last buffer in the chain, then it attempts to merge the buffers into a larger buffer when possible in order to perform large physical I/O.



The maximum chain size that the operating system can use is 64Kb. Merged buffers cannot cross a "chain boundary", which is 64Kb. This means that if the series of buffers to be merged crosses a 64Kb boundary, then the merging will be split into a maximum of 2 buffers less than 64Kb in size.

If your application performs writes to a file or performs random reads, do not increase ***max_buf_data_size*** to 64k unless the size of the read or write is 64k or greater. Smaller writes cause the data to be read into the buffer synchronously first, then the modified portion of the data will be updated in the buffer cache, then the buffer will be flushed asynchronously. The synchronous read of the data will drastically slow down performance if the buffer is not already in the cache. For random reads, JFS will attempt to read an entire buffer, causing more data to be read than necessary.

Note that buffer merging was not implemented initially on IA-64 systems (currently using JFS 3.3 on 11.22 and JFS 3.5 on 11.23). So using the default ***max_buf_data_size*** of 8Kb would result in a maximum physical I/O size of 8Kb. Buffer merging is implemented in the 11iv2 0409 release scheduled to be released in the fall of 2004.

Discovered Direct I/O

With HP OnLineJFS, direct I/O will be enabled if the read or write size is greater than or equal to the file system tunable ***discovered_direct_iosz***. The default ***discovered_direct_iosz*** is 256k. As with direct I/O, all discovered direct I/O will be synchronous. Read ahead and flush behind will not be performed with discovered direct I/O.

If read ahead and flush behind is desired for large reads and writes or the data needs to be reused from buffer cache, then the ***discovered_direct_iosz*** can be increased as needed.

Extent Allocation Policies

Since most applications use a write size of 8k or less, the first extent is often the smallest. If the file system uses mostly large files, then increasing the ***initial_extent_size*** may reduce file fragmentation by allowing the first extent allocation to be larger.

However, increasing the ***initial_extent_size*** may actually increase fragmentation if many small files (<8k) are created, as the large initial extent is allocated from a large free area, then trimmed when the file is closed.

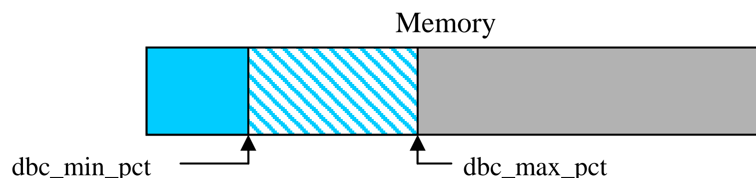
As extents are allocated, they get progressively larger (unless the file is trimmed when it is closed). Extents will grow up to ***max_seqio_extent_size*** blocks (default 2048 blocks). The ***max_seqio_extent_size*** file system tunable can be used to increase or decrease the maximum size of an extent.

System Wide Tunables

Several system wide tunables are available which can be modified to enhance performance. These tunables help control Fancy Read Ahead, buffer cache, the JFS Inode Cache, and the Directory Name Lookup Cache (DNLC).

Buffer Cache

The system tries to keep frequently accessed data in special pages in memory called the buffer cache. The size of the buffer cache can be tuned using the system wide tunables ***bufpages*** and/or ***nbuf*** for static buffer cache, and ***dbc_max_pct*** and ***dbc_min_pct*** for dynamic buffer cache. Dynamic buffer cache grows quickly as new buffers are needed. The buffer cache is slow to shrink, as memory pressure must be present in order to shrink buffer cache.



The buffer cache should be configured large enough to contain the most frequently accessed data. However, processes often read large files once (for example, during a file copy) causing more frequently accessed pages to be flushed or invalidated from the buffer cache.

The advantage of buffer cache is that frequently access data can be accessed through memory, without requiring disk access. Also, data being read from disk or written to disk can be done asynchronously.

The disadvantage of buffer cache is that data in the buffer cache may be lost during a system failure. Also, all the buffers associated with the file system must be flushed to disk or invalidated when the file system is synchronized or unmounted. A large buffer cache can cause delays during these operations.

If data needs to be accessed once without keeping the data in the cache, various options such as using direct I/O, discovered direct I/O, or the VX_SETCACHE ioctl with the VX_NOREUSE option may be used. For example, rather than using cp(1) to copy a large file, try using dd(1) instead using a large block size as follows:

```
# dd if=srcfile of=destfile bs=256k
```

By default, cp(1) reads and writes data using 64KB logical I/O. Using dd, data can be read and written using 256Kb I/Os. The large I/O size will cause dd to engage the Discovered Direct I/O feature of HP OnlineJFS and the data will be transferred using large Direct I/O. There are several advantages of using dd(1) over cp(1):

- Transfer can be done using a larger I/O transfer size
- Buffer cache is bypassed, thus leaving other more important data in the cache.
- Data is written synchronously, instead of asynchronously, avoiding large buildup of dirty buffers which can potentially cause large I/O queues or process that call sync()/fsync() to temporarily hang.

JFS Metadata Buffer Cache

The JFS Metadata Buffer Cache was introduced with JFS 3.5. Metadata is the structural information in the filesystem, and includes the superblock, inodes, directories blocks, bit maps, and the Intent Log. Prior to JFS 3.5, the data from these disk structures was read into the standard HP-UX Buffer Cache. On JFS 3.5, there is now a separate cache for the JFS metadata. The separate metadata cache allows JFS to do some special processing on metadata buffers, such as shared read locks, which can improve performance by allowing multiple readers of a single buffer.

By default, the maximum size of the cache scales depending on the size of memory. The maximum size of the JFS Metadata Cache varies depending on the amount of physical memory in the system according to the table below:

Memory Size (Mb)	JFS Metadata Cache (Kb)	JFS Metadata Cache as a percent of memory
256	32000	12.2%
512	64000	12.2%
1024	128000	12.2%
2048	256000	12.2%
8192	512000	6.1%
32768	1024000	3.0%
131072	2048000	1.5%

Table 4: Size of JFS Metadata Cache

The kernel tunable ***vx_bc_bufhwm*** specifies the maximum amount of memory in kilobytes (or high water mark) to allow for the buffer pages. By default, ***vx_bc_bufhwm*** is set to zero, which means the default maximum sized is based on the physical memory size (see Table 3).

When you move from JFS 3.3 to JFS 3.5, the metadata cache can tape up to 15% of memory (depending on the memory size) above what was taken by JFS 3.3, since the metadata on JFS 3.3 was included in the HP-UX buffer cache. If the system is already running close to the low memory threshold, the increased memory usage can potentially degrade performance.

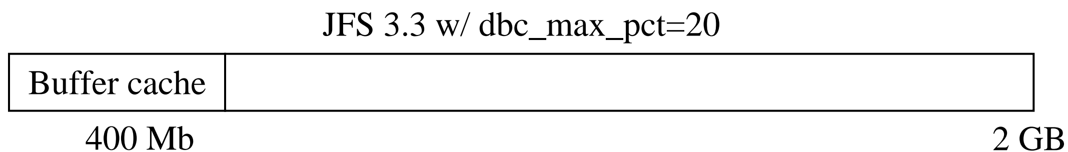


Figure 1

Figure 1 represents a typical 2GB memory system using JFS 3.3 and dbc_max_pct set to 20%. In this configuration, the HP-UX buffer cache, which also contains the metadata, takes up maximum of 400 MB of memory.

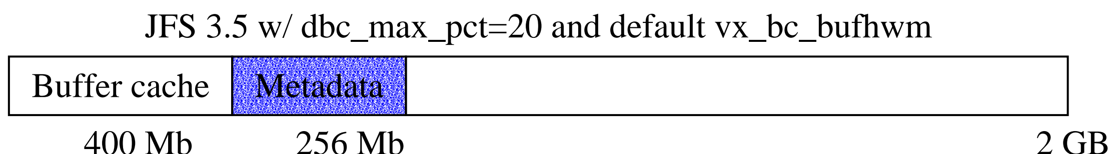


Figure 2

In Figure 2, using the same configuration, JFS 3.5 is installed. The metadata cache can now grow to a maximum of 256MB, and the HP-UX buffer cache can still grow to a maximum of 400 MB. Thus less spaces is available for other users of memory.

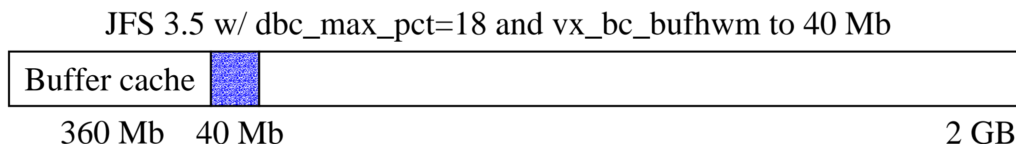


Figure 3

If you still wish for both HP-UX buffer cache and the JFS metadata cache to consume the same amount of memory, the tunables will need to be adjusted. In Figure 3, a 90/10 ration of data to metadata is used and dbc_max_pct is tuned to 18% and vx_bc_bufhwm is tuned to 40 MB. The 90/10 ration is probably good for applications that use large files, such as database applications. Applications using a lot of small files with frequent file creations/deletions or large directories can potentially use a 60/40 or 50/50 ratio.

JFS Inode Cache

JFS maintains a cache of the most recently accessed inodes in memory. The JFS inode cache is separate from the HFS inode cache. The JFS inode cache is dynamically sized. The cache grows as new inodes are accessed and contracts when old inodes are not referenced. There must be at least one inode entry in the JFS inode cache for each file that is opened at a given time.

While the inode cache is dynamically sized, there is a maximum size for the JFS inode cache. The default maximum size is based on the amount of memory present. For example, a system with 2 to 8 GB of memory will have maximum of 128,000 inodes. The maximum number of inodes can be tuned using the system wide tunable ***vx_ninode***. Most systems do not need such a large JFS inode cache. For 11.11, PHKL_24783 is needed in order to tune ***vx_ninode***.

Note also that the HFS inode cache tunable ***ninode*** has no affect on the size of the JFS inode cache. If /stand is the only HFS file system in use, ***ninode*** can be tuned lower (400, for example). However, if ***ninode*** is set low, be sure to tune ***ncsize*** or ***vx_ncsize*** on systems using JFS 3.3 or earlier to allow for a large DNLC since both JFS and HFS share the DNLC. On JFS 3.5, there is a separate DNLC for JFS.

Directory Name Lookup Cache

The Directory Name Lookup Cache (DNLC) is used to improve directory lookup performance. The DNLC contains a number of directory and file names in a cache sized by the ***ncsize*** and ***vx_ncsize*** system wide tunables. The DNLC is searched first, prior to searching the actual directories. Only filenames with less than 39 characters (JFS 3.3) or 32 characters (JFS 3.5) can be cached.

The DNLC may not help if an entire large directory cannot fit into the cache, so an ll(1) or find(1) of a large directory could push out other more useful entries in the cache. Also, the DNLC does not help when adding a new file to the directory or when searching for a non-existent directory entry.

When a file system is unmounted, all of the DNLC entries associated with the file system must be purged. If a file system has several thousands of files and the DNLC is configured to be very large, a delay could occur when the file system is unmounted.

On JFS 3.3 and prior version, the DNLC shared by HFS and JFS is sized by the ***ncsize*** and ***vx_ncsize*** system wide tunables. With JFS 3.5, there is a separate DNLC for JFS and it is sized by ***vx_ninode***.

Large Directories

One myth is that a file system with a large number of small files (for example 500,000 files which are 4kb in size or less) does not need to be defragmented. However, a file system with thousands of small files is likely to have very large directories. Directories are good examples of files that are often opened, extended, and closed. Therefore directories are usually fragmented. "Small" is a relative term. The number of blocks taken by a directory also depends on the size of each file name. As a general rule, consider a small directory to be one that has fewer than 10,000 directory entries.

When adding a new file to a directory or looking for a non-existent file, every directory block must be search. If the directory has 1000 directory blocks, then the system must do at least 1000 I/Os to add a single file to the directory or search for a non-existent file.

Simultaneous directory searches also incur contention on the inode, extent map, or directory blocks, potentially single-threading access to the directory. Long delays can be detected when doing multiple 'll' commands on a single large directory.

With JFS 3.3, if the file system is upgraded to the version 4 disk layout, then large directories can be defragmented when doing an extent reorganization, so that the directory contains larger but fewer extents. The reorganization of a directory can relieve bottlenecks on the indirect blocks when large directories are searched simultaneously.

JFS ioctl() options

Cache Advisories

While the mount options allow you to change the cache advisories on a per-file system basis, the **VX_SETCACHE** ioctl() allows you to change the cache advisory on a per-file basis. The following options are available with the VX_SETCACHE ioctl:

- VX_RANDOM - Treat read as random I/O and do not perform read ahead
- VX_SEQ - Treat read as sequential and perform maximum amount of read ahead
- VX_DIRECT - Bypass the buffer cache. All I/O to the file is synchronous. Application buffer must be word aligned and I/O must begin on a block boundary.

- VX_NOREUSE - Invalidate buffer immediately after use. This option is useful for data that is not likely to be reused.
- VX_DSYNC - Data is written synchronously, but the file's timestamps in the inode are not flushed to disk. This option is similar to using the O_DSYNC flag when opening the file.
- VX_UNBUFFERED - Same behavior as VX_DIRECT, but updating the file size in the inode is done asynchronously.

See the manpage vxfsio(7) for more information. HP OnLineJFS product is required to us the VX_SETCACHE ioctl().

Allocation Policies

The **VX_SETEXT** ioctl() passes 3 parameters: a fixed extent size to use, the amount of space to reserve for the file, and an allocation flag defined below.

- VX_NOEXTEND - write will fail if an attempt is made to extend the file past the current reservation
- VX_TRIM - trim the reservation when the last close of the file is performed
- VX_CONTIGUOUS - the reservation must be allocated in a single extent
- VX_ALIGN - all extents must be aligned on an extent-sized boundary
- VX_NORESERVE - reserve space, but do not record reservation space in the inode. If the file is closed or the system fails, the reservation is lost.
- VX_CHGSIZE - update the file size in the inode to reflect the reservation amount.

Reserving file space insures that you do not run out of space before you are done writing to the file. The overhead of allocating extents is done up front. Using a fixed extent size can also reduce fragmentation.

See the manpage vxfsio(7) for more information. HP OnLineJFS product is required to us the VX_SETEXT ioctl().

Patches

Be sure to check the latest patches for fixes to various performance related problems. Several problems have been identified and fixed with patches such as PHKL_27212 (11.0) and PHKL_27121 (11.11), such as:

- Sequential I/O if read size < 64k
- Multiple readers with Fancy Read Ahead
- Sequential I/O with Fancy Read Ahead
- Random reads
- Backward and forward

Summary

There is no single set of values for the tunable to apply to every system. You must understand how your application accesses data in the file system to decide which options and tunables can be changed to maximize the performance of your file system.

For additional reading, please refer to the "HP JFS 3.3 and HP OnLineJFS 3.3 VERITAS File System 3.3 System Administrator's Guide for HP-UX 11.0 and HP-UX 11i" and the "Veritas Filesystem 3.5 (HP OnlineJFS/JFS 3.5) Administrator's Guide" available at www.docs.hp.com.

newfs mkfs	Mount options	File system tunables	System wide tunables	Per-file attributes
bsize logsize version	Blkclear mincache convosync datainlog log delaylog tmplog logiosize tranflush	read_pref_io read_nstream write_pref_io write_nstream read_ahead max_diskq write_throttle max_buf_data_size discovered_direct_iosz initial_extent_size max_seqio_extent_size	vx_fancyra_enable vx_ninode vx_ncsize ncsize nbuf bufpages dbc_min_pct dbc_max_pct vx_bc_bufhwm	VX_SETCACHE VX_SETEXT