Common Misconfigured HP-UX Resources

By: Mark Ray, Global Solutions Engineering
Steven Albert, Global Systems Engineering
Jan Weaver, Global Systems Engineering

Overview

Physical memory is a finite resource. It is also a shared resource with many processes attempting to access this finite resource. Not only do processes need memory in order to run, the HP-UX operating system (or kernel) also needs spaces for its critical resources and tables. Some of these resources are static (do not change in size) and some are dynamic. Many of these resources can be configured to be a certain size or configured to be limited by a certain value.

While there are many possible system resources that can take up memory, this document attempts to identify some of the common misconfigured HP-UX resources and how they impact your system.

There are many reasons why an HP-UX resource may be misconfigured. The most common reason is that customer environments are unique. There is no one set of tunables that is best for all systems. Understanding how these resources are managed and how they impact memory utilization is key to a successful configuration.

This document discusses the following topics:

- The HFS Inode Cache
- The HP-UX Buffer Cache
- The JFS Inode Cache
- The JFS Metadata Buffer Cache
- Semaphores Tables

The HFS Inode Cache

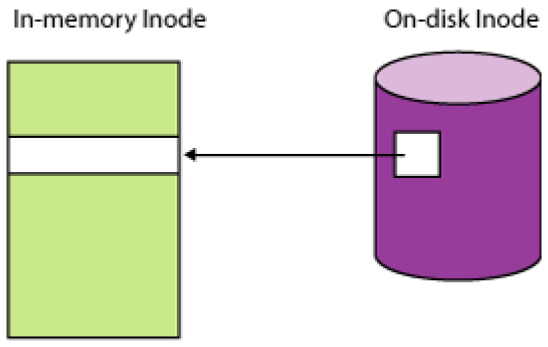
With the introduction of the Journaled File System (JFS), many systems now use the High Performance File System (HFS) for the boot file system (`/stand`) only. Since the JFS inode cache is managed separately from the HFS inode cache, you may need to adjust the size of your HFS inode cache.

This section addresses the following questions regarding the HFS inode cache:

- What is an inode cache?
- How is the HFS inode cache managed?
- How much memory is required for the HFS inode cache?
- What dependencies are there on the HFS inode cache?
- Are there any guidelines for configuring the HFS inode cache?

What is an Inode Cache?

An inode cache is simply a holding location for inodes from disk. Each inode in memory is a superset of data that contains the inode from disk. The disk inode stores information for each file, such as the file type, permissions, timestamps, size of file, number of blocks, and block map. The in-memory inode stores the on-disk inode information along with overhead used to manage the inode in memory. This information includes pointers to other structures, pointers used to maintain linked lists, the inode number, lock primitives, and other related information.



The inode cache is the collection of in-memory inodes with its various linked lists used to manage the inodes. Once the inode is brought into memory, subsequent access to the inode can be done through memory without having to read or write it to disk.

One inode cache entry must exist for every file that is opened on the system. If the inode table fills up with active inodes, the following error will be seen on the console and in the `syslog` file and the `open()` system call will fail:

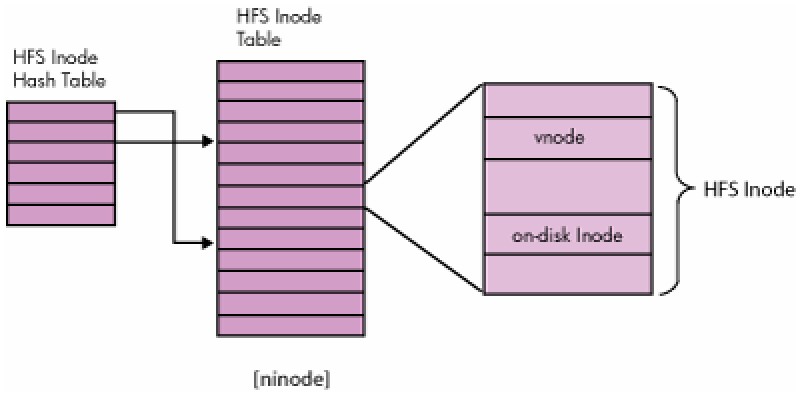
```
  inode table is full
```

Once the last close is done on a file, the inode will be put on a free list but it is not necessarily removed from the cache. The inode cache may contain some files that are closed, so if the file is reopened a disk access will not occur because the inode is already in memory.

The HFS Inode Cache is a Static Cache

The HFS inode cache is a statically sized table built during bootup. It is simply an array of in-memory HFS inodes, which is hashed for quick lookups into the table. Inodes are either in use or on the free list. It is common for the HFS inode cache to appear full since even closed files are maintained in the HFS inode cache.

The size of the cache is determined by the `ninode` tunable. Note that the tunable only sizes the HFS inode cache, and does not affect the JFS inode cache. The tunable also sizes the HFS inode hash table, which is the previous "power of 2" based on `ninode`. For example, if the `ninode` tunable is configured for 1500 inodes, then the hash table will have 1024 hash entries (since 1024 is the previous power of 2 to 1500).



When an inode lookup is performed, the device and inode number are both used to hash into the inode hash table. From the hash header, a linked list of inodes that hash to the same hash header is analyzed to see if the desired inode is found. If the desired inode is found, use the inode in memory.

If the desired inode for the file is not found, reuse the least recently used inode, which is the first inode on the free list. The inode information on disk is then read into this newly obtained inode. If there are no inodes on the free list, the system call will fail with the "inode table is full" message.

Determining the Memory Cost of the HFS Inode Cache

You can determine the cost of the HFS inode cache table by identifying the size of the inode hash table and the actual cache of inodes. Note that the inode cache is simply an array of vnode/inode structures (the vnode is the Virtual File System layer part). Since the vnode is actually part of the inode, you only need to consider the size of the inode. You can use the following table to identify the number of bytes needed per entry for each structure.

	11i v1.5	**11.0 32-bit**	**11.0 64-bit**	**11i v1 32-bit**	**11i v1 64-bit**	**11i v2**
inode	336	336	488	367	496	576
hash entry	8	8	16	8	16	16

Using the previous table, you can calculate the memory cost of the HFS inode cache. For example, if `ninode` is configured at 10,000 on an HP-UX 11i v1 64-bit system, then the system would use 4844 Kb for the HFS inode table (496*10000 / 1024), and 128 Kb for the HFS hash table (previous power of 2 (8192) * 16 / 1024).

The HFS Inode Cache and the DNLC

The `ninode` tunable used to configure the HFS inode cache size can also impact the size of the Directory Name Lookup Cache (DNLC). The size of the DNLC used to be entirely dependent on the value of `ninode`. However, this caused some problems since the DNLC was used by HFS and non-HFS file systems (such as JFS). For example, even if the /stand file system was the only HFS file system, a customer would have to configure `ninode` very large in order to get a large enough DNLC to handle the JFS files.

The dependency on the `ninode` tunable is reduced with the introduction of the following two tunables:

- ncsize — Introduced with PHKL_18335 on HP-UX 10.x. Determines the size of the Directory Name Lookup Cache independent of `ninode`.
- vx_ncsize — Introduced in HP-UX 11.0. Used with ncsize to determine the overall size of the DNLC.

On HP-UX 11i v1 and earlier, while you can tune `ncsize` independently of `ninode` the default value is still dependent on `ninode`, which is calculated as follows:

```
(NINODE+VX_NCSIZE)+(8*DNLC_HASH_LOCKS)
```

On HP-UX 11i v2 and later, the default value for `ncsize` is 8976.

Beginning with JFS 3.5 on HP-UX 11i v1, the DNLC entries for JFS files are maintained in a separate JFS DNLC, which is sized by `vx_ninode`; therefore, `vx_ncsize` can be set to 0. On HP-UX 11i v2, the `vx_ncsize` parameter has been removed.

Configuring Your HFS Inode Cache

On HP-UX 11i v1 and earlier, the default size of the HFS inode cache is based on a number of tunables, most notably `nproc`, which is calculated as follows:

```
((NPROC+16+MAXUSERS)+32+(2*NPTY)
```

On HP-UX 11i v2 and later, the default size of `ninode` is 4880.

However, some systems will need to configure a larger HFS inode cache, and some systems will need to configure a smaller cache. The first thing you need to remember is how many HFS file systems you presently have. If the boot file system (`/stand`) is your only HFS file system, then you can configure `ninode` with a very low size (maybe 200-400). If you configure a small HFS inode cache, be sure that the DNLC is configured appropriately for other file system types (such as NFS) by configuring the `ncsize` tunable.

At a minimum, you need to configure the HFS inode cache so that it is large enough to hold all of the HFS files that are open at any given instance in time. For example, you may have 200,000 HFS inodes, but only 1000 are simultaneously opened at the same time. If so, give your system some headroom and configure `ninode` to be 4000.

If you use mostly HFS files systems, the default value of `ninode` is still good for many systems. However, if the system is used as a file server, with random files opened repeatedly (for example as a Web server, mail server, or NFS server), then you may consider configuring a larger `ninode` value (perhaps 40,000-80,000). When configuring a large `ninode` cache, remember the memory resources that will need to be allocated.

The HP-UX Buffer Cache

The HP-UX buffer cache configuration can be confusing, and the HP-UX buffer cache is frequently over or under configured. Understanding how the HP-UX buffer cache is maintained and used can help you determine the proper configuration for your application environment.

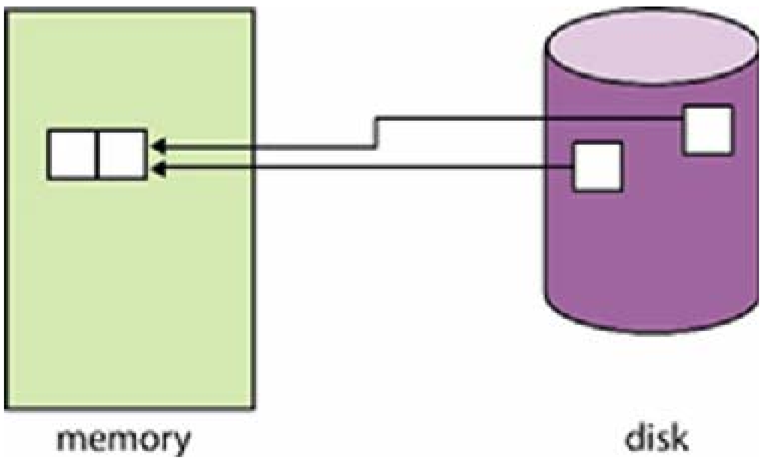
This section addresses the following questions regarding the HP-UX 11i v1 and HP-UX 11i v2 buffer cache:

- What is the buffer cache?
- How does a static buffer cache differ from a dynamic buffer cache?
- How does the buffer cache work?
- How much memory is required for the buffer cache and its related structures?
- What are the advantages and disadvantages of using the buffer cache?
- Can the buffer cache be bypassed?
- Are there any guidelines for configuring the buffer cache?

Note that the Unified File Cache (UFC) introduced in HP-UX 11i v3 replaces the traditional HP-UX buffer cache for caching of file data. He HP-UX buffer cache is still used to cache metadata (non-user file data) for non-VxFS filesystems. Therefore, this section only applies to HP-UX 11i v2 and earlier.

What is the Buffer Cache?

The buffer cache is an area of memory where pages from the secondary storage devices are stored. The buffer cache is used to reduce access to the secondary storage devices by storing frequently accessed pages in memory.



Once the file data is in memory, subsequent access can be performed in memory, without the need to access the secondary storage device.

Static Buffer Cache Versus Dynamic Buffer Cache

By default, the buffer cache is dynamic so that it can expand or shrink in size over time. A dynamic buffer cache is tuned by setting the `nbuf` and `bufpages` kernel tunable parameters to zero, and by setting the minimum and maximum ranges as a percentage of memory, `dbc_min_pct` and `dbc_max_pct` respectively. The default values are:

```
  dbc_max_pct      50
```

```
 dbc_min_pct       5
```

The dbc_min_pct tunable cannot be less than 2 and dbc_max_pct cannot be greater than 90. The dbc_min_pct and dbc_max_pct tunables are dynamic on HP-UX 11i v2 and can be modified without a system reboot.

When the system is initially booted, the system allocates dbc_min_pct (the default is 5 percent) of memory for buffer pages (each page is 4,096 bytes). The system also allocates one buffer header for every two buffer pages. The size of the buffer cache will grow as new pages are brought in from disk. The buffer cache can expand very rapidly, so that it uses the maximum percentage of memory specified by the dbc_max_pct tunable. A large file copy or a backup are operations that can cause the buffer cache to quickly reach its maximum size. While the buffer cache expands quickly, it decreases in size only when there is memory pressure.

You can configure a static buffer cache to a fixed size by setting either nbuf or bufpages. Setting nbuf specifies the number of buffer headers that should be allocated. Two buffer pages are allocated for each buffer header for a total of nbuf*2 buffer pages. If the bufpages kernel parameter is set and nbuf is 0, then the number of buffer pages is set to bufpages and one buffer header is allocated for every two buffer pages for a total of bufpages/2 buffer headers. If both nbuf and bufpages are set, then nbuf is used to size the buffer cache. You can also configure a static buffer cache by setting the dbc_min_pct and dbc_max_pct tunables to the same value.

There are trade-offs associated with either a static or dynamic buffer cache. If memory pressure exists, a static buffer cache cannot be reduced, potentially causing more important pages to be swapped out or processes deactivated. In contrast, some overhead exists in managing the dynamic buffer cache, such as the dynamic allocation of the buffers and managing the buffer cache address map or buffer cache virtual bitmap (the bufmap and bcvmap tunables are discussed later in more detail). A dynamic buffer cache also expands very rapidly, but contracts only when memory pressure exists.

How the Buffer Cache Works

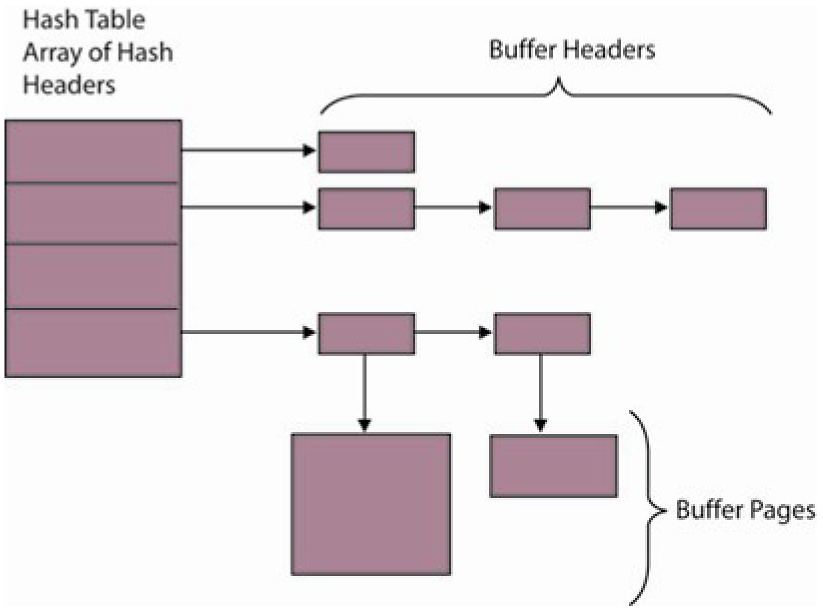
The main parts of the buffer cache are the buffer cache hash table, buffer headers, and buffer pages themselves. At a minimum, allocate one page for a buffer header, even if the buffer is only 1 KB in size. The maximum buffer size is 64 Kb. Every buffer in the cache is linked through buffer cache hash lists. When data from a disk is needed, use the block device (specifically the vnode address of the block device) and the block number to calculate a hash index into the buffer cache hash table, which is an array of buffer cache hash headers. The buffer cache hash header will point to a linked list of buffers whose block device and block number hash to the same hash header. By searching the hash chain, we can tell if the requested block exists in the buffer cache. In HP-UX 11i v1 and later, there is one hash lock to cover every eight hash chains. Prior to HP-UX 11i v1 there was a fixed number of 128 hash locks.

If an attempt is made to access a block from a device and it does not exist in the appropriate hash chain, a buffer cache miss occurs and one of two actions will occur:

- A new buffer is allocated (if a dynamic buffer cache is used). Data must be read in from disk.
- An existing buffer is reused (if a static buffer cache is used or the buffer cache is already at the dbc_max_pct) and data must be read in from disk. The buffer reused is either a buffer that has been invalidated (for example, a file has been removed or a file system has been unmounted), or the buffer has not been accessed recently.

However, if the desired buffer is found in the buffer cache, then the data can be accessed without accessing the disk.

The following figure shows the HP-UX buffer cache.



Note that buffers remain in the buffer cache even after a file is closed. Thus, if a file is reopened a short time later, the buffers may still be available in the buffer cache. For example, if your buffer cache is 500 MB in size, and you enter a `grep` command on a 100-MB file, each data block will need to be read into the buffer cache as the file is scanned. However, if you enter a subsequent `grep` command on the same file, the file should be accessed via the buffer cache without accessing the disk device even though the file was closed after the first `grep` command.

Buffer Cache and Memory

How much memory does the buffer cache take? This seems like a simple question. However, if you configure your buffer cache to a dynamic maximum of 10 percent of physical memory on a system with 12 GB of memory, then the maximum size of the buffer cache is 1.2 GB of memory. Note that this only represents the buffer pages. Other structures used to manage the buffer cache are not accounted for in this total. These other structures include the following:

- Buffer headers
- Buffer cache hash table
- Buffer hash locks
- Buffer cache address map and buffer cache virtual map

Buffer Headers

Each buffer in the buffer cache needs a header structure that defines what the block represents, how it can be used, and how it is linked. For a 64-bit system, buffer headers are 600 bytes on HP-UX 11.11 and 692 bytes on HP-UX 11i v2. If the buffer cache is fixed, `nbuf` buffer headers are allocated at system initialization. If the buffer cache is dynamic, buffer headers are allocated dynamically as needed. When more buffer headers are needed a page of memory is allocated and carved up into as many buffer headers as will fit in one 4-KB page.

Buffer Cache Hash Table

Blocks in the buffer cache are hashed so that they can be accessed quickly. The number of hash entries is computed at boot time and is one-quarter of the number of free memory pages rounded up to the nearest power of two. Therefore, a system with 12 GB of memory will have approximately one million hash table entries regardless of the buffer cache configuration. A page is 4,096 bytes, thus 12 GB represents 3,145,728 pages. One-quarter of that is 786,432 rounded up to the next power of two, which is 1,048,576. Each hash header is 40 bytes on HP-UX 11i v1 and 32 bytes on HP-UX 11i v2, therefore, a 12-GB system would have 40 MB of memory allocated to the buffer cache hash table on HP-UX 11i v1 and 32 MB on HP-UX 11i v2.

Buffer Cache Hash Locks

Instead of having a lock for each buffer cache hash header, a single lock is used for multiple hash headers. This reduces the amount of memory needed for hash locks. Prior to HP-UX 11i v1 there were 128 hash locks. Therefore, the number of rows each hash lock covered increased as the size of the buffer cache increased.

The following table specifies how many hash chains each hash lock covers. On HP-UX 11i v1 and higher, the number of hash table rows covered by each lock is fixed at 8.

System Memory Size	Hash Table Size	Hash Entries Per Lock (HP-UX 11.0)	Hash Entries Per Lock (HP-UX 11I v1, HP-UX 11i v2)	Total Hash Table Memory
1 GB	65536	512	8	2.5 MB
2 GB	131072	1024	8	5 MB
4 GB	262144	2048	8	10 MB
8 GB	524288	4096	8	20 MB
12 GB	1048576	8192	8	40 MB
32 GB	2097152	16384	8	80 MB
256 GB	16777216	131072	8	640 MB

Buffer Cache Address Map and Buffer Cache Virtual Map

In HPUX 11.00 and earlier, the buffer cache address map (there are actually two maps: `bufmap` and `bufmap2`) is a resource map used to keep track of virtual addresses used by the buffer cache. The `bufmap` contains entries for each address range that is free and available for use by the buffer cache. The `bufmap` varies in size based on the memory size. It takes approximately 1 percent of memory on a 32-bit system and 2 percent of memory on a 64-bit system.

The buffer cache virtual map (`bcvmap`) is a bitmap introduced with HP-UX 11i v1 that represents pages in the buffer cache. Since it is a bitmap, its memory requirements are smaller than the `bufmap`. By default, the bitmap is sized (in bits) to the number of physical memory pages multiplied by the `bcvmap_size_factor` kernel tunable (the default is 2). There is some overhead to manage the bitmap groups, but overall the memory usage is insignificant.

The default `bcvmap_size_factor` value of 2 is fine for many systems, especially those that use a `dbc_max_pct` of 20 or less. However, when a dynamic buffer cache is used, buffers of varying sizes can be allocated and deallocated over time. The allocation and deallocation of variable-sized buffers can fragment the `bcvmap`. If there are not any bitmap areas available to represent the size of the buffers needed, the system may thrash. This is more common with systems where `dbc_max_pct` is configured at 50 percent or more or where different sized buffers are used. Buffers of different sizes come into play when multiple HFS file systems of different block sizes are used, or multiple JFS file systems are mounted with different `max_buf_data_size` settings, or with NFS file systems mounted with a read/write size other than 8 K. In such caches, increase the `bcvmap_size_factor` parameter to at least 16.

Refer to the *vxtunefs*(1M) manpage for more information on the `max_buf_data_size` parameter.

The `bcvmap_size_factor` parameter is only available on 64-bit HP-UX 11i v1 systems that have the PHKL_27808 patch installed. This parameter is also available with HP-UX 11i v2.

The following table shows the memory usage of the buffer pages and buffer cache headers for HP-UX 11.0 with a buffer header size of 600 bytes:

System Memory Size	**10% Buf Pages/Buf Headers**	**20% Buf Pages/Buf Headers**	**50% Buf Pages/Buf Headers**
1 GB	100 MB/7.3MB	200 MB/15 MB	500 MB/36 MB
2 GB	200 MB/15MB	400 MB/30 MB	1 GB/75 MB
4 GB	400 MB/30MB	800 MB/60 MB	2 GB/150 MB
8 GB	800 MB/60 MB	1.6 GB/120 MB	4 GB/300 MB
12 GB	1.2 GB/90 MB	2.4 GB/180 MB	6 GB/450 MB
32 GB	3.2 GB/240 MB	6.4 GB/480 MB	16 GB/1.2 GB
256 GB	25.6 GB/2 GB	51 GB/4 GB	128 GB/9.4 GB

The following table shows the memory usage of buffer pages and buffer cache headers for HP-UX 11i v1 and HP-UX 11i v2 with a buffer header size of 692 bytes:

System Memory Size	**10% Buf Pages/Buf Headers**	**20% Buf Pages/Buf Headers**	**50% Buf Pages/Buf Headers**
1 GB	100 MB/8.4 MB	200 MB/16.9 MB	500 MB/42 MB
2 GB	200 MB/16.9 MB	400 MB/33.8 MB	1 GB/86.5 MB
4 GB	400 MB/33.8 MB	800 MB/67.6 MB	2 GB/173 MB
8 GB	800 MB/67.6 MB	1.6 GB/138.4 MB	4 GB/346 MB
12 GB	1.2 GB/103.8 MB	2.4 GB/207.6 MB	6 GB/519 MB
32 GB	3.2 GB/276.8 MB	6.4 GB/553.6 MB	16 GB/1.35 GB
256 GB	25.6 GB/2.2 GB	51 GB/4.3 GB	128 GB/10.8 GB

Advantages of Using the Buffer Cache

There are several advantages to using buffer cache, including:

- **Small sequential I/O**
  Applications read data from the file system in various size requests, which may not line up to the actual file system block size. Without the buffer cache, each request would have to go to the physical disk for the entire file system block, even though only a few bytes might be needed. If the next read is from the same physical disk block, it would have to be read in again since it was not saved. However, with the buffer cache, the first read causes a physical I/O to the disk, but subsequent reads in the same block are satisfied out of the buffer cache.

- **Read ahead**
  If file system access is generally sequential, the buffer cache provides enhanced performance via read ahead. When the file system detects sequential access to a file, it begins doing asynchronous reads on subsequent blocks so that the data is already available in the buffer cache when the application requests it.

  For HFS file systems, general sequential reads are configured via the `hfs_ra_per_disk`

  system tunable. If you are using LVM striping, multiply the `hfs_ra_per_disk` value by the number of stripes.

  For JFS 3.1, the initial read ahead starts out small and, as the sequential access continues, JFS reads ahead more aggressively.

  For JFS 3.3 and later, the read-ahead range is the product of the `read_pref_io` and `read_nstream` parameters. When sequential access is first detected, four ranges are read into the buffer cache (4 `* read_pref_io * read_nstream`). When an application finishes reading in a range, a subsequent range is prefetched. The read-ahead size can greatly benefit sequential file access. However, applications that generally do random I/O may inadvertently trigger the large read ahead by occasionally reading sequential blocks. This read-ahead data will likely be unused due to the overall random nature of the reads.

  For JFS 3.3 and later, you can control the size of the read-ahead data with the `vxtunefs` `read_nstream` and `read_pref_io` parameters; for JFS 3.5/4.1, you can turn the read ahead size off by setting the `vxtunefs` parameter `read_ahead` to 0. For JFS 3.1, you cannot tune the read-ahead size.
- **Hot blocks**
  If a file system block is repeatedly accessed by the application (either a single process or multiple processes), then the block will stay in the buffer cache and can be used without having to go to the disk each time the data is needed. The buffer cache is particularly helpful when the application repeatedly searches a large directory, perhaps to create a temporary file. The directory blocks will likely be in buffer cache if they are frequently used and physical disk access will not be required.
- **Delayed writes**
  The buffer cache lets applications perform delayed or asynchronous writes. An application can write the data to the buffer cache and the system call will return without waiting for the I/O to complete. The buffer will be flushed to disk later using commands such as `syncer`, `sync`, or `fsync`. Performing delayed writes is sometimes referred to as write behind.

Disadvantages of Using the Buffer Cache

While it may seem that every application would benefit from using the buffer cache, using the buffer cache does have some costs, including:

- **Memory**
  Depending on how it is configured, the buffer cache may be the largest single user of memory. By default, a system with 8 GB of memory may use as much as 4 GB for buffer cache pages alone (with `dbc_max_pct` set to 50). Even with a dynamic buffer cache, a large buffer cache can contribute to overall memory pressure. Remember that the buffer cache will not return the buffer pages unless there is memory pressure. Once memory pressure is present, buffer pages are aged and stolen by `vhand`. Under memory pressure, buffer cache pages are stolen at a rate three times that of user pages.

- **Flushing the buffer cache: the syncer program**
  The syncer program is the process that flushes delayed write buffers to the physical disk. Naturally, the larger the buffer cache, the more work that must be done by the syncer program. The HP-UX 11.0 syncer is single threaded. It wakes up periodically and sequentially scans the hash table for blocks that need to be written to the physical device. The default syncer interval is 30 seconds, which means that every 30 seconds the entire buffer cache hash table is scanned for delayed write blocks. The syncer program runs five times during the syncer interval, scanning one-fifth of the buffer cache each time.

  HP-UX 11i v1 and HP-UX 11i v2 are more efficient at this in that the syncer program is multithreaded with one thread per CPU. Each CPU has its own dirty list and each syncer thread is responsible for flushing buffers from its own dirty list. This improves buffer cache scaling in that only dirty buffers are scanned and each thread has its own list preventing contention around a single list.

- **Other sync operations**
  Various system operations require that dirty blocks in the buffer cache be written to disk before the operation completes. Examples of such operations are the last close on a file, or an unmount of a file system, or a sync system call. These operations are independent of the syncer program and must traverse the entire buffer cache looking for blocks that need to be written to the device. Once the operation completes, it must traverse the buffer cache hash table again, invalidating the buffers that were flushed. These traversals of the buffer cache hash chains can take time, particularly if there is contention around the hash lock.

- **IO throttling**
  Besides just walking the hash chains and locking/unlocking the hash locks, the larger the buffer cache, the larger the number of dirty buffers that are likely to be in the cache needing to be flushed to the disk. This can cause large amounts of write I/O to be queued to the disk during sync operations. A read request could get delayed behind the writes and cause an application delay. Flushes of the buffer cache can be throttled, limiting the number of buffers that can be enqueued to a disk at one time. By default, throttling is turned off. For JFS 3.1, you can enable throttling if the PHKL_27070 patch is installed by setting vx_nothrottle to 0. This alleviates read starvation at the cost of sync operations such as unmounting a file system. For JFS 3.3 and later, you can control the amount of data flushed to a disk during sync operations via the vxtunefs max_diskq parameter.

- **Write throttling**
  Setting max_diskq to throttle the flushing of dirty buffers has a disadvantage. Processes that perform sync operations, such as umount or bdf, can stall since the writes are throttled. Setting max_diskq does not prevent applications from continuing to perform asynchronous writes. If writes to large files are being done, it is possible to exhaust all the buffers with dirty buffers, which can delay reads or writes from other critical applications.

  With JFS 3.5, a new tunable was introduced: write_throttle. This controls the number of dirty buffers a single file can have outstanding. If an application attempts to write faster than the data can be written to disk and the write_throttle amount has been reached, the application will wait until some of the data is written to disk and the amount of data in the dirty buffers falls back below the write_throttle amount.

- **Large I/O**
  The maximum size of a buffer page is 64 KB. For I/O requests larger than 64 KB, the request must be broken down into multiple 64-KB I/O requests. Therefore, reading 256 KB from disk

may require four I/O requests. However, if the buffer cache is bypassed, a single 256 KB direct I/O could potentially be performed.

- **Data accessed once**
  Management of the buffer cache requires additional code and processing. For data that is accessed only once, the buffer cache does not provide any benefit for keeping the data in the cache. In fact, by caching data that is accessed only once, the system may need to remove buffer pages that are more frequently accessed.

- **System crash**
  Since many writes are delayed, the system may have many dirty buffers in the buffer cache that need to be posted to disk when the system crashes. Data in the buffer cache that is not flushed before the system comes down is lost.

Bypassing the Buffer Cache

There are several ways that I/O can avoid using the buffer cache altogether. Bypassing the buffer cache is known as direct I/O.

For the following JFS features, the HP Online JFS license is needed to perform direct I/O:

- `mincache=direct, convosync=direct`

  For JFS file systems, if the file system is mounted with the `mincache=direct` option, the `convosync=direct` option, or both, then reads and writes bypass the buffer cache and go directly to the user-supplied buffer. If a file system is mounted with these options, all I/O to the file system bypasses the buffer cache. Read ahead and write behind are not available since there is no intermediate holding area. Refer to the *mount_vxfs*(1M) manpage for more information.

- `ioctl (fd,VX_SETCACHE, VX_DIRECT)`

  For JFS file systems, this `ioctl` call will set the access for the file referenced by `fd` as direct and will bypass the buffer cache. This call applies only to the instance of the file represented by `fd`. Other applications opening the same file are not affected. Refer to the *vxfsio*(7) manpage for more information.

- Discovered direct I/O

  JFS provides a feature called discovered direct IO, where I/O requests larger than a certain size are done using direct I/O. Large I/O requests are typically performed by applications that read data once, such as backup or copy operations. Since the data is accessed once, there is no benefit to caching the data. Caching the data may even be detrimental as more useful buffers may get flushed out to make room for this once-accessed data. Therefore, large I/O requests on JFS file systems are performed directly and bypass the buffer cache. For JFS 3.1, the discovered direct I/O size is fixed at 128 Kb. For JFS 3.3 and later, the default discovered direct I/O size is 256 Kb, but can set with `vxtunefs` command by setting the `discovered_direct_iosz` tunable.

- Raw I/O

  If access to the data is through the raw device file, then the buffer cache is not used.

- Async I/O

Some databases use the `async` driver (`/dev/async`), which performs asynchronous I/O, but bypasses the buffer cache and reads directly into shared memory segments.

Be careful not to mix buffered I/O and direct I/O. This results in increased overhead to keep the direct and buffered data in sync.

Buffer Cache Guidelines

Providing general guidelines for tuning the buffer cache is very difficult. So much depends on the application mix that is running on the system, but some generalizations can be made.

If you are using a database, the database buffering will likely be more efficient than the system buffering. The database is more likely to understand the I/O patterns and keep any relevant buffers in memory. Given a choice, memory should be assigned to the database global area rather than the system buffer cache.

HP-UX 11 v1 is more efficient at handling large buffer caches than HP-UX 11.0. The term “large” is relative, but for this discussion consider a buffer cache larger than 1 GB or greater than 50 percent of memory to be large. In general, the buffer cache on 11.0 should be configured to 1 GB or less due to scaling issues with large caches, but you can increase this size on HP-UX 11i v1 and HP-UX 11i v2. However, if you are using a large buffer cache on HP-UX 11i v1 you should have PHKL_27808 or any superseding patch installed to increase the buffer cache virtual map size.

The larger the buffer cache, the longer `sync` operations will take. This particularly affects file system mount and unmount times. If file systems need to be mounted or unmounted quickly (for example during an Serviceguard package switch), then a smaller buffer cache is better.

If the buffer cache is configured too small, the system could be constantly searching for available buffers. The buffer cache should probably be configured to a minimum of 200 MB on most systems.

Applications that benefit most from large caches are often file servers, such as NFS or Web servers, where large amounts of data are frequently accessed. Some database applications that do not manage their own file access may also fall into this category. Please check with your application vendor for any vendor-specific recommendations.

The JFS Inode Cache

HP-UX has long maintained a static cache in physical memory for storing High Performance File System (HFS) file information (inodes). The VERITAS File System (HP OnlineJFS/JFS) manages its own cache of file system inodes which will be referred to here as the JFS inode cache. The JFS inode cache is managed much differently than the HFS inode cache. Understanding how the JFS inode cache is managed is key to understanding how to best tune the JFS inode cache for your unique environment. Improper tuning of the JFS inode cache can affect memory usage or inode lookup performance.

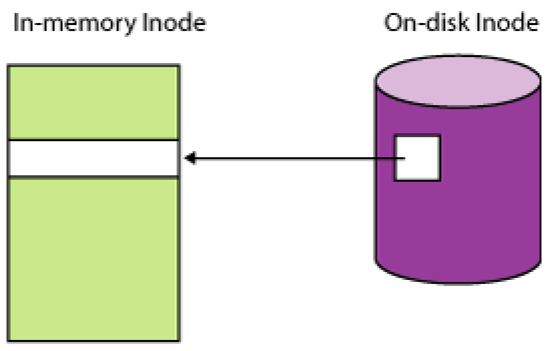
This section will address the following questions regarding the JFS inode cache:

- What is an inode cache?
- What is the maximum size of the inode cache?
- How can I determine the number of inodes in the cache?
- How can I determine the number of inodes in use?
- How does JFS manage the inode cache?
- How much memory is required for the JFS inode cache?
- Are there any guidelines for configuring the JFS inode cache?

What is an Inode Cache?

An inode cache is a holding location for inodes from disk. Each inode in memory is a superset of data, which contains the inode from disk. The disk inode stores information for each file such as the file type, permissions, timestamps, size of file, number of blocks, and extent map. The in-memory inode stores the on-disk inode information along with overhead used to manage the inode in memory. This information includes pointers to other structures, pointers used to maintain linked lists, the inode number, lock primitives, and so forth.

Once the inode is brought into memory, subsequent access to the inode can be done through memory without having to read or write it to disk.



One inode cache entry must exist for every file that is opened on the system. If the inode table fills up with active inodes, the following error will be seen on the console and in the `syslog` file and the `open()` system call will fail:

```
  vx_iget - inode table overflow
```

Once the last close is done on a file, the inode will be put on a free list but it is not necessarily removed from the cache. The inode cache will likely contain some files that are closed, so if the file is reopened a disk access will not occur as the inode is already in memory.

The JFS Inode Cache is a Dynamic Cache

Unlike the High Performance File System (HFS), the JFS inode cache is a dynamic cache. A dynamic cache is a cache that grows and shrinks based on need. As files are opened, the number of inodes in the JFS inode cache grows. As files are closed, they are moved to a free list and can be reused at a later time. However, if the inode is inactive for a certain period of time, the inode is freed and space is return to the kernel memory allocator. Over time, the numbers of inodes in the inode cache will grow and shrink.

Maximum Inodes in the JFS Inode Cache

While the JFS inode cache is dynamically sized, there is still an absolute maximum number of inodes that can be maintained in the inode cache. The following table shows the default maximum number of inodes in the JFS inode cache[1]:

Physical Memory	**Default Maximum Inodes for JFS 3.3 and above**
256 Mb	16,000
512 Mb	32,000
1 GB	64,000
2 GB	128,000
8 GB	256,000
32 GB	512,000
128 GB	1,024,000

Specific values exist for memory sizes less than 256 Mb, but the sizes are not mentioned here since most HP-UX systems should be using 256 Mb of memory or more.   For systems equipped with memory cells that can be configured as removable, memory in the above table refers to *kernel available* memory and does not include removable memory.

If the size of memory falls in between the memory sizes listed in the previous table, a value proportional to the two surrounding values is used. For example, a system using JFS 3.3 with 5 GB of memory would have a default maximum number of inodes in the inode cache of 192,000.

Note the default maximum number of inodes seems very high. Remember that it is a maximum, and the JFS inode cache is dynamic, as it can shrink and grow as inodes are opened and closed. The maximum size must be large enough to handle the maximum number of concurrently opened files at any given time, or the “vx_iget - inode table overflow” error will occur.

For JFS 3.3, you can identify the maximum number of JFS inodes that can be in the JFS inode cache using the following adb command:

```
 # echo “vxfs_ninode/D” | adb -k /stand/vmunix /dev/mem
```

1. JFS 3.3 is supported on 11i v1; JFS 3.5 is supported on HP-UX 11i v1 and 11i v2; and JFS 4.1 is supported on HP-UX 11i v2, HP-UX 11i v3; and JFS 5.0 is supported on HP-UX 11i v2, HP-UX 11i v3 (planned).

If you are using JFS 3.5 and above, you can use the `vxfsstat` command to display the maximum number of inodes in the inode cache as follows:

```
# vxfsstat / | grep inodes
     3087 inodes current     128002 peak              128000 maximum
   255019 inodes alloced     251932 freed

# vxfsstat -v /  | grep maxino
vxi_icache_maxino             128000    vxi_icache_peakino         128002
```

Note that in the previous example the inode cache can handle a maximum of 128,000 inodes.

Determining the Current Number of Inodes in the JFS Inode Cache

Determining how many inodes are currently in the inode cache is difficult on JFS versions prior to JFS 3.5 as existing user tools do not give this information. You can use the following `adb` command to display the current number of inodes in the JFS inode cache if using JFS 3.3:

```
# echo “vx_cur_inodes/D” | adb –k /stand/vmunix /dev/mem
```

Using JFS 3.5 and above, you can use the `vxfsstat` command to display the current number of inodes in the inode cache as follows:

```
# vxfsstat / | grep inodes
     3087 inodes current     128002 peak              128000 maximum
   255019 inodes alloced     251932 freed

# vxfsstat -v /  | grep curino
vxi_icache_curino               3087    vxi_icache_inuseino           635
```

Note from the previous output that the current number of inodes in the cache is 3087.

Determining the Number of Active JFS Inodes in Use

While a number of the JFS inodes exist in the cache, remember that not all of the inodes are actually in use as inactive inodes exist in the cache.

For JFS 3.3, there is no easy method to determine the actual inodes in use. For JFS 3.5 and above, we can again use `vxfsstat` to determine the actual number of JFS inodes that are in use:

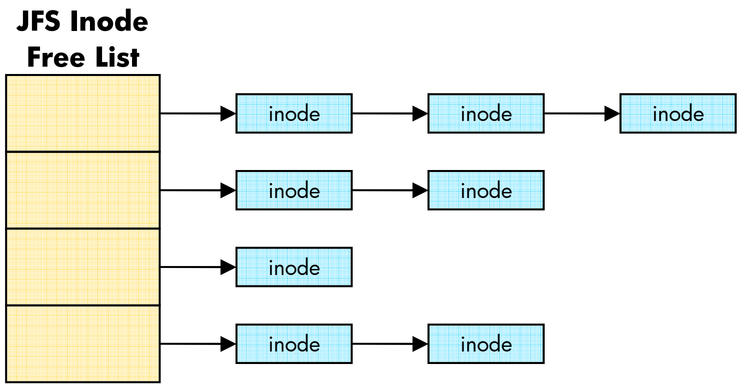
```
# vxfsstat -v / | grep inuse
vxi_icache_curino             128001    vxi_icache_inuseino          635
```

The inode cache is filled with 128,001 inodes but only 635 are in use. The remaining inodes are inactive, and if they remain inactive one of the `vxfsd` daemon threads will start freeing the inodes after a certain period of time.

Note that the current number of inodes (128,001) is greater than the maximum number of inodes (128,000). This behavior is normal as there are few exceptions that allow allocating a few additional inodes in the cache.

The Inode Free Lists

When a file is not accessed or closed, it is placed on one of many free lists. The free lists store inodes that can be readily reused. To avoid lock contention when maintaining the free lists, there are actually many free list headers. The diagram below illustrates this concept:



JFS uses a number of factors to determine the number of free lists on the system, such number of inodes, maximum number of processors supported, etc. The number of free lists is not tunable and can vary from one JFS version to the next. The number of free lists can be identified with the following adb command:

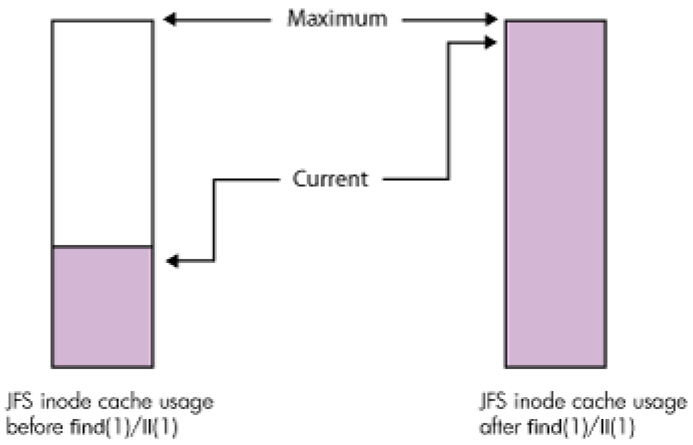
```
# echo "vx_nfreelists/D" | adb -k /stand/vmunix /dev/mem      # 11.11
# echo "vx_nfreelists/D" | adb -o /stand/vmunix /dev/kmem     # 11.23
```

More discussion on how the free lists affect memory utilization will follow.

Growing the JFS Inode Cache

When a file is opened or accessed and it does not already exist in the cache, its inode must be brought in from disk into the JFS inode cache. JFS must make a decision as to whether to use one of the existing inodes on a free list, or to allocate a new inode if you have not allocated the maximum number of inodes.   For JFS 3.3 and later, an inode must be on a free list for three minutes before it is reused. If the inodes on the free list have been there for less than two or three minutes, then JFS will allocate more inodes from the kernel memory allocator as necessary.

Consider an application that does a `stat` system call on many files in a very short timeframe. Example applications or processes are the `find`, `ls`, `ll`, or backup commands. These commands traverse through the file systems at a very rapid pace. If you have enough files on your system, you can easily fill up the JFS inode cache in a very short time. However, the files are usually not kept open. Therefore, if the JFS inode cache fills up, an inode is claimed off the free list even though it has not been on the free list for the appropriate amount of time.


Maximum
Current
JFS inode cache usage before find(1)/ll(1)
JFS inode cache usage after find(1)/ll(1)

Note that maintaining all these inodes may or may not be helpful. For example, if the inode cache can hold 128,000 inodes, and the *find*(1) command traverses 129,000 inodes, then only the last

128,000 inodes would be in the cache. The first 1000 inodes would have been reused when reading in the last 1000 inodes. If you use the `find` command again, it would have to recache all 129,000 inodes.

However, if the `find` command only traversed through 127,000 inodes, then all of the inodes would be in the cache for the second `find` command, which would then run much faster.

As an example, consider an HP-UX 11i v1 system using JFS 3.5. Prior to a find command, the vxf`sstat` command shows the current number of inodes to be 3087:

```
# vxfsstat / | grep inodes
      3087 inodes current     128002 peak                 128000 maximum
   255019 inodes alloced     251932 freed
```

When you enter the `find` command, the inode cache will fill up if it traverses enough files:

```
# find / -name testfile
# vxfsstat / | grep inodes
   128001 inodes current     128002 peak                 128000 maximum
   379936 inodes alloced     251935 freed
```

After entering the `find` command, note that the number of inodes currently in the cache jumped to 128,001.

The “peak” value of 128,002 represents the highest value that the “inodes current” has been since the system was booted. Again, it is normal for the “current” and “peak” counts to be greater than the “maximum” value by a few inodes.

Shrinking the JFS Inode Cache

Periodically, one of the JFS daemon (`vxfsd`) threads runs to scan the free lists to see if any inodes have been inactive for a period of time. If so, the daemon thread begins to free the inodes back to the kernel memory allocator so the memory can be used in future kernel allocations.

The length of time an inode can stay on the free list before it is freed back to the kernel allocator and the rate at which the inodes are freed varies depending on the JFS version as shown in the following table:

	JFS 3.3	**JFS 3.5/JFS 4.1/JFS 5.1**
Minimum time on free list before being freed (seconds)	500	1800
Maximum inodes to free per second	50	10 - 25

For example, JFS 3.3 will take approximately 2000 seconds or 33 minutes to free up 100,000 inodes.

With JFS 3.5 and above, you can see the minimum time on the free list before the inode is a candidate to free using the `vxfsstat` command:

```
# vxfsstat -i / | grep "sec free"
     1800 sec free age
```

This value is also considered the `ifree_timelag`, which you can also display with `vxfsstat` on JFS 3.5:

```
# vxfsstat -v / | grep ifree
vxi_icache_recycleage          1035    vxi_ifree_timelag            1800
```

Consider the JFS 3.5 case again. The system begins to free the inodes if they have been inactive for 30 minutes (1800 seconds) or more. About 30 minutes after entering the find command, the inodes start to free up:

```
# date; vxfsstat -v / | grep -i curino
Thu May  8 16:34:43 MDT 2003
vxi_icache_curino            127526    vxi_icache_inuseino           635
```

The `vxfsstat` command is executed again 134 seconds later:

```
  # date; vxfsstat -v / | grep -i curino
  Thu May  8 16:36:57 MDT 2003
  vxi_icache_curino            127101    vxi_icache_inuseino         635
```

Note that 425 inodes were freed in 134 seconds, about 3 inodes per second.

After being idle all evening, the next day, the number of inodes in the inode cache were reduced down again as all the inactive files were freed:

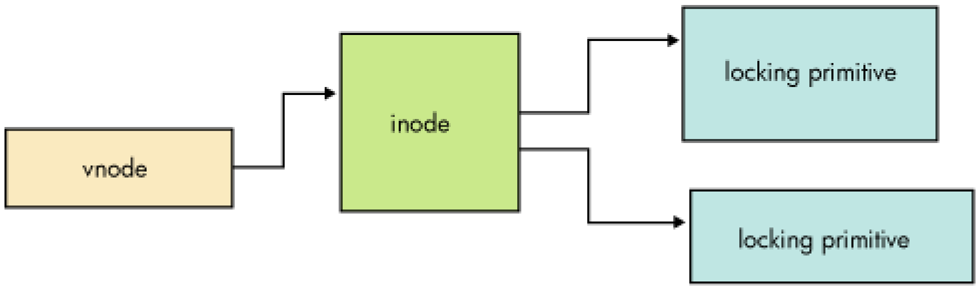
```
  # date; vxfsstat -v / | grep -i curino
  Fri May  9 14:45:31 MDT 2003
vxi_icache_curino              3011    vxi_icache_inuseino           636
```

Determining the Memory Cost Associated with each JFS Inode

The size of the individual inodes varies depending on the release. Prior to JFS 3.3, the vnode (virtual file system node) and inode were allocated together as a single resource. On JFS 3.3 and above, the vnode is allocated separately from the inode. On a 32-bit operating system, the inode is smaller as the pointer fields are only four bytes. On a 64-bit operating system, the pointer fields are eight bytes.

The kernel memory allocator also impacts the amount of space used when allocating the inodes and associated data structures. You can only allocate memory using certain sizes. For example, if the kernel tries to allocate an 80-byte structure, the allocator may round the size up to 128 bytes, so that all allocations in the same memory page are of the same size.

For each inode, you also need to account for locking structures that are also allocated separately from the inode.



You can use the following table to estimate the memory cost of each JFS inode in the inode cache (measured in bytes). Each item reflects the size as allocated by the kernel memory allocator:

Structures	**JFS 3.3 11.11 32-bit**	**JFS 3.3 11.11 64-bit**	**JFS 3.5 11.11 64-bit**	**JFS 3.5 11.23**	**JFS 4.1 11.23**	**JFS 5.1 11.23**	**JFS 4.1 11.31**
inode	1024	1364	1364	1364	1490	1490	1490
vnode	128	184	184	248	248	248	376
locks	272	384	352	96	96	120	240
Total	1352	1902	1850	1708	1834	1858	2106

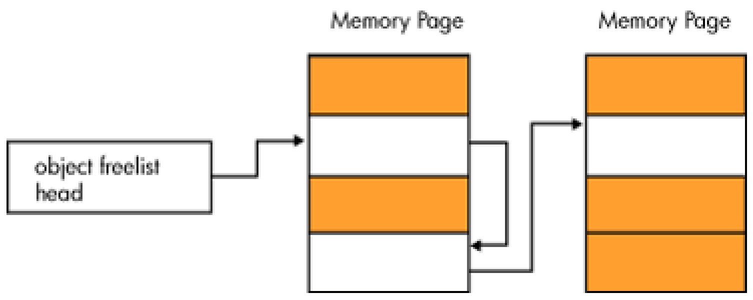
Note that the previous table lists a minimal set of memory requirements. There are also other supporting structures, such as hash headers and free list headers. Other features may use more memory. For example, using Fancy Readahead on JFS 3.3 on a file will consume approximately 1024 additional bytes per inode. Access control lists (ACLs) and quotas can also take up additional space, as well as Cluster File System information.

For example, consider an HP-UX 11i v3 system using JFS 4.1 that has 2 GB of memory. If you enter an `ll` or `find` command on a file system with a large number of files (greater than 128,000), then the inode cache is likely to fill up. Based on the default JFS inode cache size of 128000, the minimum memory cost would be approximately 256 MB or 12 percent of total memory.

However, if you then add more memory so the system has 8 GB of memory instead of 2 GB, then the memory cost for the JFS inode will increase to approximately 512 MB. While the total memory cost increases, the percentage of overall memory used for the JFS inode cache drops to 6.25 percent.

Effects of the Kernel Memory Allocator

The internals of the kernel memory allocator have changed over time, but the concept remains the same. When the kernel requests dynamic memory, it allocates an entire page (4096 bytes) and then subdivides the memory into equal sized chunks known as “objects”. The kernel does allocate pages using different sized objects. The allocator then maintains free lists based on the object size.

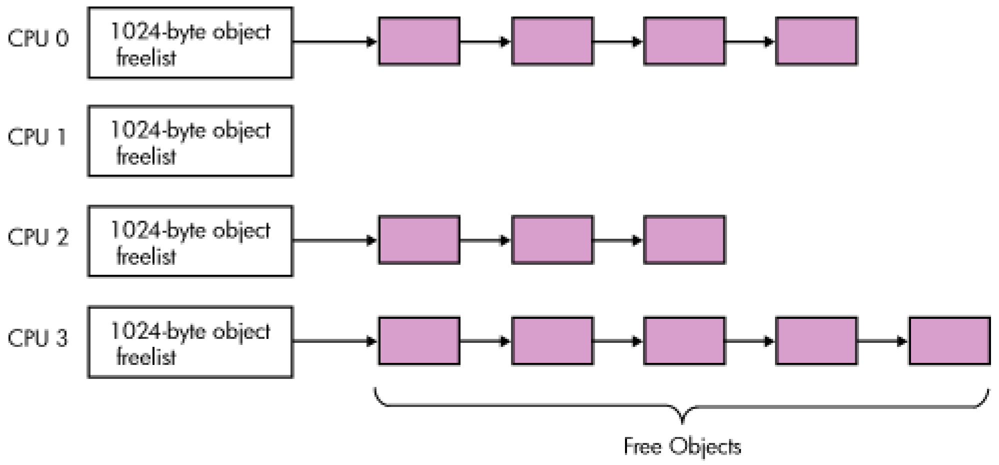


In the previous example, the memory pages are divided into four objects. In some implementations, there is some page overhead and object overhead associated with the object. For this example, assume that the size of each object is 1024 bytes. Each page may have both used and freed objects associated with it. All of the free pages are linked to a linked list pointed to by an object free list head. There is typically one object free list head associated with each CPU on the system for each object size. Therefore, CPU 0 can have a object free list for the 32-byte objects, the 64-byte objects, and so on. CPU 1 would also have corresponding object free lists.

There are multiple object sizes available but not all sizes are represented. For example, on HP-UX 11.0 there are pages that use the 1024-byte object and the 2048-byte object, but nothing in between. If JFS requests 1040 bytes of memory, then an entire 2048-byte object is taken from a page divided into two 2048-byte objects.

As inodes are allocated, the kernel will allocate memory pages and divide them into objects, which are then used by the JFS subsystem for inodes and associated data structures. To allocate 128,000 inodes that each take 1024 bytes, the system needs to allocate 32,000 pages (4 inodes per page).

As discussed earlier, the JFS inode cache is dynamic. Inodes that are not accessed are eventually freed. These freed inodes go back to the object freelist. If you enter an *ll*(1) or *find*(1) command to access 128,000 inodes, and the inodes are unreferenced for some time, then large object freelist chains can form as the JFS daemon thread starts to free the inodes. This can occur for each CPU object freelist since inodes are freed to the CPU's object chain where they were allocated.


CPU 0
1024-byte object freelist
CPU 1
1024-byte object freelist
CPU 2
1024-byte object freelist
CPU 3
1024-byte object freelist
Free Objects

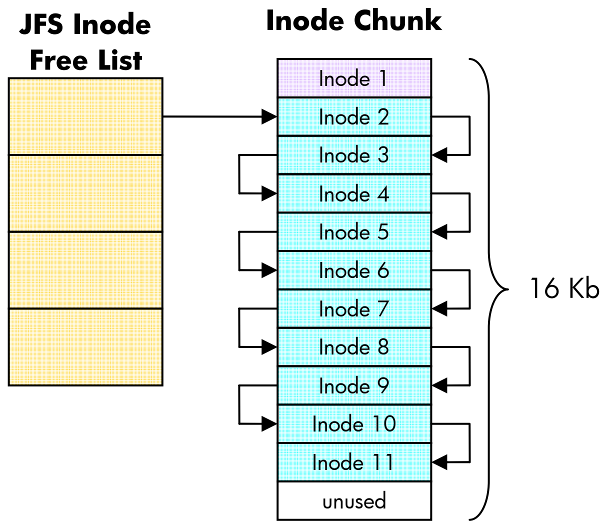
Note in the previous figure that if CPU 3 needed a new inode, then the kernel memory allocator would return the first object from the object freelist chain. However, the object freelist chain for CPU 1 is empty. If you enter an `ll` or `find` command, new inodes will be needed in the JFS inode cache. New memory pages will be allocated and divided up into objects and placed in the object freelist for CPU 1. The kernel memory allocator will not steal free objects from another CPU's free object chain. By not stealing objects, the system realizes a performance gain by reducing contention on the object chain locks. However, when an inode is freed, the kernel memory allocator will place the freed area on the free object chain for CPU 1 to be used for a subsequent allocation. The freeing of the inodes creates potentially large free object chains for each CPU as the JFS inode cache shrinks.

While the object freelists can be used for other 1024-byte allocations, they cannot be used for other sized allocations as all the objects allocated from a page must be the same size. If you have four CPUs, then you can potentially consume memory for freed inodes on the object freelist as well as used inodes in the JFS inode cache.

The HP-UX kernel performs "object coalescing", such that if all the objects in a given page are freed, then the memory page can be returned to the free pool and allocated for use in other areas (process memory, different size memory objects, and others). However, this "object coalescing" only occurs if there is memory pressure present and there is some overhead in performing the coalescing.

Inode Allocation with JFS 4.1 and later

Beginning with JFS 4.1, the inode algorithms change to allocate inodes more efficiently. Rather than allocating one 4 Kb page which is used to allocate JFS inodes, a larger "chunk" of memory was used. The inode allocation chunk size is 16Kb. Currently, JFS can carve out 11 inodes out of a 16 Kb chunk.



So on JFS 4.1 or later, when a new inode is needed and the JFS inode cache can be expanded; JFS will allocate a chunk of inodes, return one inode to the requestor and place the remaining 10 inodes on one of the inode free lists. Thus the burden of the inode coalescing is now on JFS instead of the kernel memory allocator. In order to free a chuck back to the kernel memory allocator, every inode in the chunk must be free.

Allocating inodes in chunk does have a disadvantage. Note that all inodes in a chunk that are free must be on the same free list. In order for JFS to free inodes back to the kernel memory allocator, every inode in the chunk must be free. Also, when trying to find an inode on the free list to re-use, JFS will look at a given free list first, and if there are no inodes on the free list, it must steal inodes from another free list. However, unlike earlier version of JFS, since JFS 4.1 allocates inodes in chunks, an entire chunk of free inodes must be available on another free list in order to steal inodes from another free list. Thus with JFS 4.1, it is less likely that inodes will be moved from one free list to another. Thus we need a sufficient number of inodes on each list. Tuning the maximum number of inodes on each free list will be discussed in the next 2 sections.

Tuning the Maximum Size of the JFS Inode Cache on JFS 3.5 or earlier

Every customer environment is unique. The advantage to a large inode cache is that you do not have to access the disk every time the inode is needed. If you are doing continuous random lookups on a directory with 64,000 files in it (such as opening a file or using a `stat` system call), then having a large cache is helpful. However, if you are only using a `find` or `ll` command occasionally on a set of 512,000 inodes, then having 128,000 inodes in the cache will not help. You need to understand how your files are accessed to know whether or not a large inode cache will be helpful. The memory cost associated with each inode should be taken into account as well.

You can tune the maximum size of the JFS inode cache using the `vx_ninode` tunable. With JFS 4.1 on HP-UX 11i v2, `vx_ninode` can be tuned dynamically using `kctune`.

At a minimum, you must have at least one JFS inode cache entry for each file that is opened at any given time on your system. If you are concerned about the amount of memory that JFS can potentially take, then try to tune `vx_ninode` down so that the cache only takes about1-2 percent of overall memory. Most systems will work fine with `vx_ninode` tuned to20,000-50,000. However, you need to consider how many processes are running on the system and how many files each process will have open on average. Systems used as file servers and Web servers may have performance benefits from using a large JFS inode cache and the defaults are sufficient.

Note that tuning `ninode` does not affect the JFS inode cache as the JFS inode cache is maintained separately from the HFS inode cache. If your only HFS file system is `/stand`, then `ninode` can usually be tuned to a low value (for example, 400).

Tuning the Maximum Size of the JFS Inode Cache on JFS 4.1 or later

The introduction of allocating inodes in chunks has added a new dynamic to the tuning of vx_ninode. On previous JFS versions, it was safe to tune the vx_ninode down to a smaller value, such as 20,000 on some systems. Now the number of free lists becomes a major factor, especially on large memory systems (>8GB of physical memory) which have a large number of free lists. For example, if you have 1000 free lists and vx_ninode is tuned to 20,000, then there are only 20 inodes allocated per free list. Note that 20 is an average. Some free lists will have 11 and some free lists will have 22 as the inodes are allocated in chunks of 11. Since JFS inodes are less likely moved from one free list to another, the possibility of prematurely running out of inodes is greater when too few inodes are spread over a large number of free lists. As a general rule of thumb, you would like to have 250 inodes per free list. However, if memory pressure is an issue, then vx_ninode can be tuned down to 100 inodes per free list. This will greatly limit the ability to reduce vx_ninode. So before tuning down vx_ninode to a smaller value, be sure to check the number of free lists on the system using the following adb command for JFS 4.1 (available on 11.23 and 11.31).

```
 # echo “vx_nfreelists/D” | adb –o /stand/vmunix /dev/kmem
```

Tuning Your System to Use a Static JFS Inode Cache

By default, the JFS inode cache is dynamic in size. It grows and shrinks as needed. However, since the inodes are freed to the kernel memory allocator’s free object chain, the memory may not be available for use for other reasons (except for other same-sized memory allocations). The freed inodes on the object freelists are still considered “used” system memory. Also, the massive kernel memory allocations and subsequent frees add additional overhead to the kernel.

The dynamic nature of the JFS inode cache does not provide much benefit. This may change in the future as the kernel memory allocator continues to evolve. However, using a statically sized JFS inode cache has the advantage of keeping inodes in the cache longer and reducing overhead of continued kernel memory allocations and frees. Instead, the unused inodes are retained on the JFS inode cache freelist chains. If you need to bring a new JFS inode in from disk, use the oldest inactive inode. Using a static JFS inode cache also avoids the long kernel memory object free chains for each CPU. Another benefit to a static JFS inode cache is that the `vxfsd` daemon will not use as much CPU. On large-memory systems, `vxfsd` can use a considerable amount of CPU, reducing the size of the JFS inode cache.

Beginning with HP-UX 11i v2 (with JFS 3.5 and above) a new system-wide tunable `vxfs_ifree_timelag` was introduced to vary the length of time an inode stayed in the cache before it is considered for removal. Setting `vxfs_ifree_timelag` to -1 effectively makes the JFS inode cache a static cache. Setting `vxfs_ifree_timelag` is especially useful on large memory systems so the vxfsd daemon does not use too much CPU. The following example uses kctune to change vxfs_ifree_timelag without a reboot:

```
# kctune vxfs_ifree_timelag=-1
             :                              :
Tunable                      Value  Expression  Changes
vxfs_ifree_timelag  (before)     0  Default     Immed
                    (now)       -1  -1
```

Summary

Deciding whether or not to tune the JFS inode cache depends on how you plan to use your system. Memory is a finite resource, and a system manager needs to decide how much of the system's memory needs to be spent on a specific resource.

By default, the JFS inode cache is configured to be very large. You should understand the advantages and disadvantages of using the default JFS inode cache sizes and behavior. Consider the type of access to the files on the system and the memory cost associated with the JFS inode cache when deciding whether or not to tune the JFS inode cache.

For most customers, tuning the JFS inode cache down would save memory and potentially enhance performance by leaving more memory for other operations. With the introduction of *vxfsstat*(1M) on JFS 3.5, you can determine how many inodes are actually in use at a given time to judge how large your inode cache should be. Most systems do not need more than 20,000 inodes, so reducing the size of the JFS inode cache can reduce system memory utilization. However, be careful of reducing the size of the cache on JFS 4.1 or later by checking the number of inode free lists.

Since the JFS inode cache often expands to the maximum amount, using a fixed-sized inode cache will help the kernel memory allocator from managing large per-CPU freelists. Therefore, tuning vx_noifree or vxfs_ifree_timelag can also reduce overall system memory utilization as well as reducing the amount of CPU utilization used by the vxfsd daemon.

However, if the primary use is for the system is as a fileserver, where random file lookups are being performed constantly or sequential lookups are done and the working set is less than the size of the inode cache, then using the default JFS inode cache sizes is probably best. Application performance could be degraded if the application relies on having a larger working set of inodes in the cache.

The JFS Metadata Buffer Cache

In past releases of the VERITAS File System (HP OnlineJFS/JFS) prior to Journaled File System (JFS) 3.5, the metadata of a file system was cached in the standard HP-UX buffer cache with all of the user file data. Beginning with JFS 3.5 introduced on HP-UX 11i v1, the JFS metadata was moved to a special buffer cache known as the JFS metadata buffer cache (or metadata cache). This cache is managed separately from the HP-UX buffer cache. This metadata cache serves the same purpose as the HP-UX buffer cache, but enhancements were made to increase performance due to the unique ways the metadata is accessed.

This section will address the following questions regarding the metadata cache:

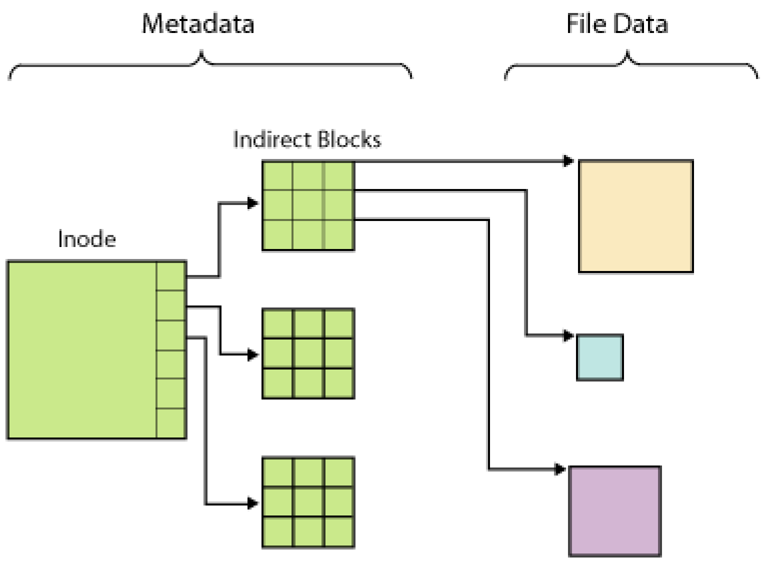
- What is metadata?
- Is the metadata cache static or dynamic?
- How much memory is required for the metadata cache?
- How can the metadata cache be tuned?
- Are there any guidelines for configuring the metadata cache?

What is Metadata?

Metadata is structural information from disk such as inodes, indirect block maps, bitmaps, and summaries.

If you consider an actual file on disk, it is made up of the inode and data blocks, and potentially indirect blocks. The inode contains an extent map to either the data blocks or other extent maps known as indirect blocks.

When inodes are first read in from disk, the file system reads in an entire block of inodes from disk into the metadata cache (similar to reading a data file). Then, the inodes that are actually being accessed will be brought into the JFS inode cache. Note the difference between inodes in the metadata cache, which contains only the disk copy, and the inode in the JFS inode cache, which contains the linked lists for hashing and free lists, the vnode, locking structures, and the on-disk copy of the inode.



The Metadata Cache: Dynamic or Static?

The metadata cache is a dynamic buffer cache, which means it can expand and shrink over time. It normally expands during periods of heavy metadata activity, especially with operations that traverse a large number of inodes, such as a `find` or `backup` command. Simply reading a large file may fill up the HP-UX buffer cache, but not the metadata cache.

Unlike the HP-UX buffer cache, which contracts only when memory pressure is present, the metadata cache contracts after the metadata buffers have been inactive for a period of time (about one hour). While the metadata cache contracts, it contracts at a slow rate, so it takes some time for inactive buffers to be reclaimed by the memory subsystem.

You can view the current size of the buffer cache and the maximum size of the buffer cache by using the `vxfsstat` command. For example:

```
# vxfsstat -b /

12:55:26.640  Thu Jan 3 2006 -- absolute sample

buffer cache statistics
   348416 Kbyte current      356040 maximum
   122861 lookups            98.78% hit rate
     2428 sec recycle age [not limited by maximum]
```

Note that the maximum amount only accounts for the buffer pages, and does not include the buffer headers, which account for approximately 22 percent more memory.

The Metadata Cache and Memory

The default maximum size of the JFS metadata cache varies depending on the amount of physical memory in the system according to the following table:

Memory Size (Mb)	**JFS Metadata Cache (Kb)**	**JFS Metadata Cache as a percent of physical memory**
512	64,000	12.2%
1024	128,000	12.2%
2048	256,000	12.2%
8192	512,000	6.1%
32768	1,024,000	3.05%
131072	2,048,000	1.53%

If the size of memory falls between two ranges, then the maximum sizes is calculated in proportion to the two neighboring memory ranges. For example, if the system has 4 GB of memory (4096 Mb), then the calculated maximum metadata cache size is 356,040 Kb, a value proportional to the 2-GB and 8-GB range.

For systems equipped with memory cells that can be configured as removable, the size of the JFS Metadata Buffer Cache cannot be more than 25% of kernel available memory.

Note that the table represents the *maximum* size of the metadata cache, which includes buffer headers and buffer pages. The cache is dynamic so it will expand as the metadata is accessed and contract slowly. Operations that touch a lot of metadata, such as a system backup, will attempt to bring all the metadata into the cache. However, it is possible that there is not enough metadata to fill the cache, even if all the metadata is brought into the cache. Use the `vxfsstat` command to see how much metadata is in the cache and what the maximum size is for buffer pages.

Note that memory usage from the table accounts for both buffer pages and buffer headers. It does not include other overhead for managing the metadata cache, such as the hash headers and free list headers. The buffer headers are dynamically allocated and deleted as needed and the size of the buffers may vary, so the actual number of buffer headers will vary as well. However, the total amount of memory for the buffers and buffer headers cannot exceed the predefined maximum size.

Tuning the Metadata Cache

The `vx_bc_bufhwm` kernel tunable specifies the maximum amount of memory in kilobytes (or high water mark) to allow for the buffer pages and buffer headers. You can set this value the with `sam` or `kmtune/kctune` command, or in the `/stand/system` file. By default, `vx_bc_bufhwm` is set to 0, which means to use the default maximum size based on the physical memory size (see the previous table).

For VERITAS journaled file system (VxFS) 3.5, the system must be rebooted after changing `vx_bc_bufhwm` for the new value to take effect. With VxFS 4.1 on 11i v2, you can immediately set `vx_bc_bufhwm` without a reboot using the `kctune` command.

When using `vxfsstat` to view the current size of the metadata buffer cache, only the buffer pages are counted. For example, consider the following `vxfsstat` output for a system with 4 GB of main memory:

```
  # vxfsstat -b / | grep current
     74752 Kbyte current     414384 maximum
```

In the previous example, 74,752 Kb is used for the buffer pages. Up to 22 percent of additional space will be used for the buffer headers (or 16,446 Kb for this example). Note, however, that the memory used for buffer pages cannot exceed 414,384 maximum, and the total amount of memory for both buffer pages and buffer headers cannot exceed the maximum size of the metadata cache, or the `vxfs_bc_bufhwm` (in this example, 507,576 Kb).

Autotuning with OL* additions and deletions

Beginning with VxFS 4.1 on HP-UX 11i v3, when cells are added or deleted with cell-based systems, the vx_bc_bufhwm is autotuned if vx_bc_bufhwm was originally set with the default value of 0. However, the JFS Metadata Cache cannot take more than 25% of the kernel available memory. For example, if you have a system with 2GB of base kernel memory and use OL* to add an additional 6 GB, then vx_bc_bufhwm would be autotuned from 256,000 Kb to 512,000 Kb. If another additional 4GB of memory is added, vx_bc_bufhwm will remain at 512,000 Kb since the JFS Metadata Buffer Cache cannot exceed 25% of kernel available memory (2GB in this case).

If you manually tune vx_bc_bufhwm to be a specific size, vx_bc_bufhwm will not be adjusted if memory additions or deletions are done, and you may need to adjust vx_bc_bufhwm if an different sized JFS Metadata Buffer Cache is needed after the OL* addition or deletion.

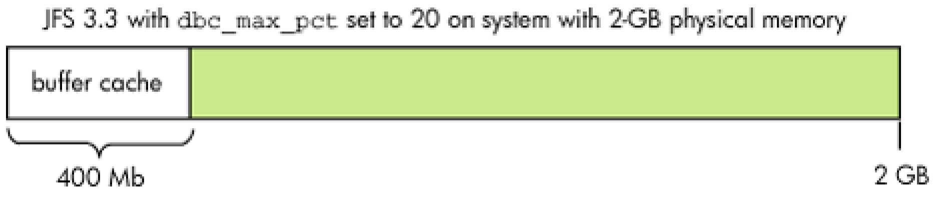
Recommended Guidelines for tuning the JFS Metadata Buffer Cache

If you upgrade from JFS 3.3 to JFS 3.5 on HP-UX 11i v1, then the metadata cache can take up to 12.2 percent of memory, depending on memory size and the amount of metadata accessed, above what was taken by JFS 3.3 since the metadata on JFS 3.3 was included in the HP-UX buffer cache. The new metadata cache can potentially increase the performance of metadata-intensive applications (for example, applications that perform a high number of file creations/deletions or those that use large directories).

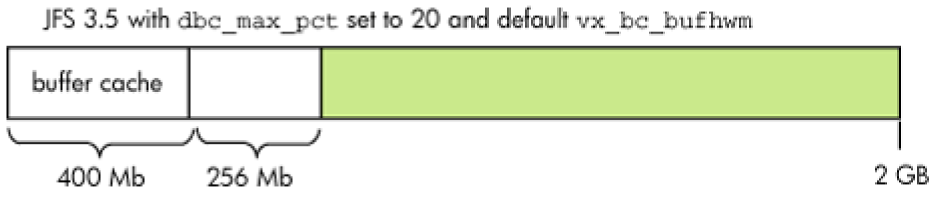
However, the memory cost must be considered. If the system is already running close to the low memory threshold, the increased memory usage can consume memory that could potentially be used for other applications, potentially degrading the performance of other applications.

Note that file systems with a large number of small files can have much more metadata than a larger file system with a smaller amount of large files. There is no way to predict how much metadata will be brought into the cache.

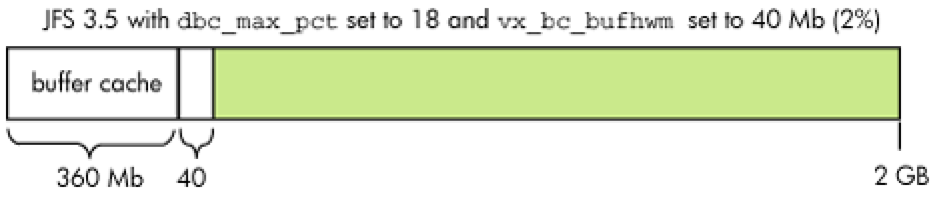
Consider the following example, a 2-GB system with `dbc_max_pct` parameter set to 20 percent running JFS 3.3 on 11i v1:


JFS 3.3 with dbc_max_pct set to 20 on system with 2-GB physical memory
buffer cache
400 Mb
2 GB

The JFS file system is then upgraded to JFS 3.5. The `dbc_max_pct` parameter is still 20 percent, but the metadata buffer cache has an additional 12 percent of memory. If the metadata cache is fully used, then less space exists for other applications. If you still desire to use only 20 percent for the buffer cache for both data and metadata, then the tunables must be evaluated and changed.



As a suggestion, consider the maximum amount of data and metadata that is desired, and then consider the amount of metadata as a percentage of overall data. Going back to the previous example, if the system should use a maximum of 20 percent of memory for both data and metadata, and you desire a 90/10 ratio of data to metadata, then change `dbc_max_pct` to 18 and `vx_bc_bufhwm` to 40,000 (or 40 Mb which is 2 percent of the 2 GB physical memory).


JFS 3.5 with dbc_max_pct set to 18 and vx_bc_bufhwm set to 40 Mb (2%)
buffer cache
360 Mb
40
2 GB

Note that this example uses a 90/10 ratio of data to metadata. The ratio you choose may be different depending on your application usage. The 90/10 ratio is probably good for applications that use large files, such as database applications. Applications that use lots of small files with frequent file creations/deletions or large directories, such as file servers, may need more space for metadata, so a 60/40 or 50/50 ratio may be appropriate.

Semaphores Tables

Many third-party applications, databases in particular, make extensive use of the semaphores (commonly referred to as System V IPC) available in HP-UX 11i v1 and higher. The installation guides of third party products often recommend changing the tunables associated with semaphores. The purpose of this document is to describe the memory impacts that are caused by changing these tunables.

This section addresses the following questions about semaphores:

- What is a semaphore?
- What interfaces are used to access semaphores?
- What tunables affect semaphore memory usage?
- How much memory is used to manage the semaphore tables?
- Are there any guidelines for configuring the semaphore tables?

What is a Semaphore?

Although this document will not go into the details of implementation, it is important to understand what a semaphore is, how it is used, and what the various tunables actually control.

In simple terms, a semaphore is a construct that is used to control access to a set of resources. It has a count associated with it that determines how many resources are available. Resources are allocated by decrementing the count and returned by incrementing the count. All updates to the semaphore are done atomically through system calls. When all resources have been given out, the next process that attempts to allocate one will go to sleep until a resource is freed, at which point it will be awakened and granted access.

A simple example will suffice to make this clear. Suppose we have three stations that can serve us. We can control access by having 3 tokens. The first person takes a token, then the second person, then the third person. Each of these people can be served simultaneously. If a fourth person wishes to be served, that person must wait until a token becomes available. When any person finishes, the token is returned and the next person in line can take it. Of course, the degenerate case is when the count is set to 1, which is how one could implement exclusive access. This is the basic idea and is the mechanism databases frequently use to control concurrent access.

Interfaces

There are three main interfaces that you can use to manage semaphores: `semget()`, `semctl()`, and `semop()`. The `semget()`routine allocates a set of semaphores. You can allocate more than one semaphore at a time and operations can be conducted on these sets. A successful call to `semget()` will return a semaphore identifier. The `semop()`routine operates on a set of semaphores. Finally, the `semctl()` routine performs control operations on the semaphore set. For more details, refer to the manpages for these calls.

Tunables

Below are the tunables related to semaphores, which are described in the following table:

Tunable	Description
`sema`	Enable or disable System V IPC semaphores at boot time
`semaem`	Maximum cumulative value changes per System V IPC semop() call
`semmni`	Number of System V IPC system-wide semaphore identifiers
`semmns`	Number of System V IPC system-wide semaphores
`semmnu`	Maximum number of processes that can have undo operations pending
`semmsl`	Maximum number of System V IPC semaphores per identifier
`semume`	Maximum number of System V IPC undo entries per process
`semvmx`	Maximum value of any single System V IPC semaphore

You can find links to these section 5 tunables on the HP documentation website at http://www.docs.hp.com.

Of the tunables described in the previous table, the `semaem` and `semvmx` tunables represent limits associated with the values in the semaphores. These tunables do not affect semaphore memory consumption. The other tunables, such as semmni, semmns, semmnu, semmsl and semume, are related to memory utilization. The tunable `sema` is obsolete as of HP-UX 11i v2

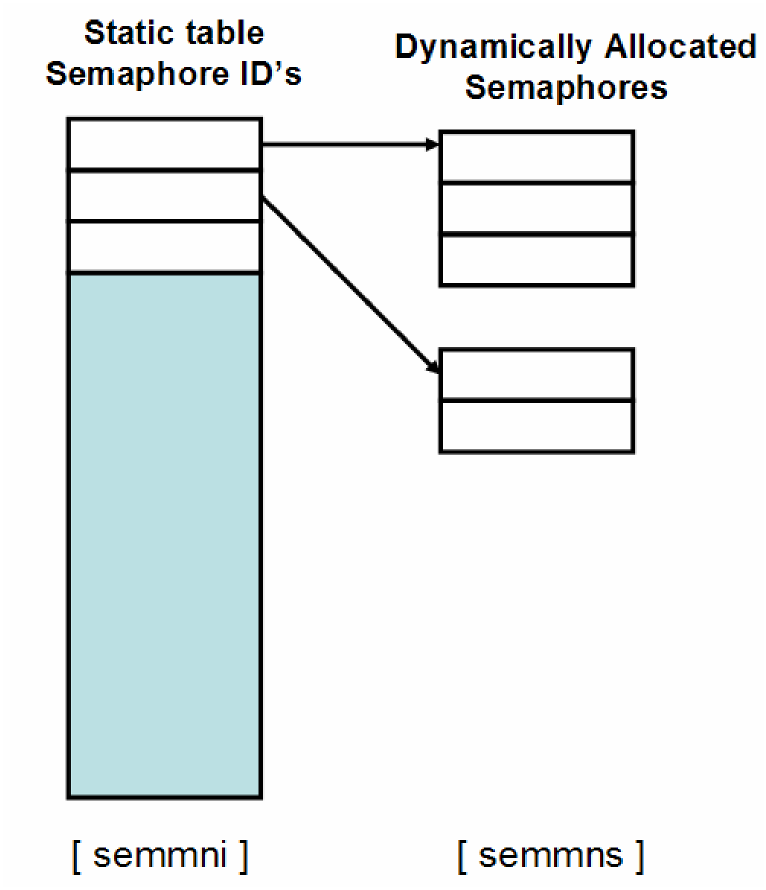
The Semaphore Tables

Until recently, the semaphore tables were statically sized, meaning that entire tables were allocated in memory when the system booted. Semaphores are allocated in groups of one or more from the semaphore table using the `semget()` system call. The `semget()` system call returned a semaphore ID from the semaphore ID table, which was then used to access the defined group of semaphores.

However, beginning with PHKL_26183 on HP-UX 11i v1, the semaphore entries that make up the semaphore table were dynamically allocated. By dynamically allocating semaphores, the system was able to reduce memory usage previously caused by allocated but unused semaphores. It can also resolve the fragmentation issues that could occur in the semaphore table. Other tables such as the semaphore undo table remained static.

The other significant change is related to the semaphore table itself. The `semmns` tunable now represents a limit to the number of semaphores that can be allocated in total. They are allocated as needed and freed when not in use. The caveat here is that the boot time table is still allocated even though it is not used (in HP-UX 11i v1). As a result, you allocate space for `semmns` semaphores and allocate space at run time for the semaphores that are currently in use. One other notable change was put into PHKL_28703 for HP-UX 11i v1, which lifts the restriction of 32,767 on `semmns`. With this patch installed, the new limit is 1,000,000.

The following figure shows the relationship between the semaphore ID table and the semaphore per semaphore identifier.



The `semmni` tunable sizes the semaphore ID table that maintains the system-wide semaphore identifiers. Each entry in the array is 96 bytes, so each increment to `semmni` will increase the boot time kernel size by 96 bytes.

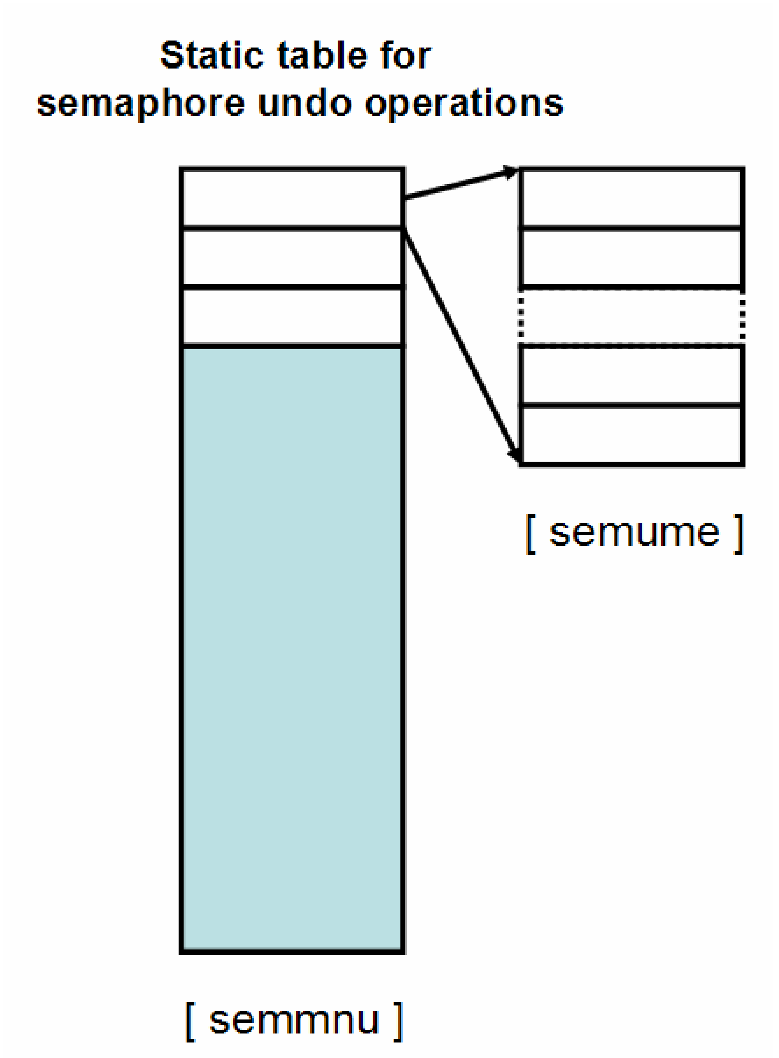
The `semmsl` tunable defines the maximum number of semaphores per semaphore identifier. It does not size a specific static table. Each semaphore is 8 bytes.

The `semmns` tunable keeps track of the system-wide semaphores. Semaphores are allocated dynamically based on requested number of semaphores through `semget(2)` call and up to the limit of `semmns` tunable value across the system.

The semaphore undo table is slightly more complex. When a process terminates because of abort or signal, the effects of a semaphore operation may need to be undone to prevent potential applications hangs. The system needs to maintain a list of pending undo operations on a per-process basis.

The `semume` tunable specifies the number of semaphore undo operations pending for each process. Each semaphore undo structure is 8 bytes in size, and there are 24 bytes of overhead for each process. As a result increasing `semume` increases the semaphore undo table by 8 bytes per process.

The `semmnu` tunable specifies the number of processes that can have semaphore undo operations pending. Increasing `semmnu` increases the size of the semaphore undo table based on the value of `semume`.



Use the following formula to calculate the size of the system semaphore undo table:

```
((24+(8*semume)) * semmnu)
```

Keep this formula in mind when changing these tunables since the effects on the boot time kernel size are multiplicative. How many undo structures does one need? There is no single answer to that, but it is important to note that undo structures are only allocated if the application specifies the `SEM_UNDO` flag. Thus, it is not necessary to bump these values unless the application installation guides instruct you to do so.

How can one go astray? Here is a real-life example. A customer tuned `semmni` to 10,000 and decided to also set `semume` and `semmnu` to 10,000. The first change was not significant, consuming only 80,000 bytes. However, the second set of changes resulted in the allocation of ((24+(8*10000))*10000) or 800,240,000 bytes! This was clearly not what the customer really wanted.

The following table is a summary of memory usage based on the semaphore tunables:

Kernel Table	**Tunable**	**Element Size**	**Default Setting**
Semaphore IDs	`semmni`	88 bytes	2048
Semaphore table	`semmns`	8 bytes	4096
-	`semmsl`	-	2048

Semaphore undo table	`semmnu`	24 bytes + (8*`semume`)	256 (`semmnu`) 100 (`semume`)

Changes in HP-UX 11i v2

Beginning with the HP-UX 11i v2, the dynamic changes described in the previous section define the default behavior; therefore, no patch is necessary. The `sema` tunable has been obsoleted. The sizes of the data structures remain the same, so all of the formulas for calculating the kernel boot size remain the same. Keep in mind that the semaphores themselves (limited by `semmns`) are allocated as needed and will not consume space at boot time. Thus, memory usage will grow and shrink as semaphores are allocated and freed. The other data structures are still allocated at boot time; therefore, you must be careful when changing `semmnu` and `semume`. One last change is that the limit on the total number of semaphores in the system has been enlarged. The new limit is 335,534,080, as described on the manpage.

Guidelines for Configuring Semaphores

Many third-party applications make heavy use of the Sys V IPC semaphore mechanism available within HP-UX. Understanding all of the tunables, what they represent, and how changing them can affect the memory consumed by your kernel is important so that you do not mistakenly tune the semaphores too high, leaving too little memory to accomplish other important tasks.

One of the most common mistakes is the setting of the semaphore undo table. Remember that there is a multiplicative effect from setting **`semmnu`** and **`semume`**.

Also, if you manage multiple systems, be sure that the semaphore tables are scaled appropriately given the memory size of each. A given set of tunables may work great on a system with 8 GB of memory, but have severe memory implications on a system with only 1 GB of memory.

For more information

http://docs.hp.com
HP technical documentation Web site.