

Dynamically Tuning the JFS Cache for Your Job

Sjoerd Visser



The purpose of this presentation is the explanation of:

IBM JFS goals: **Where was Journaled File System (JFS) designed for?**

JFS cache design: **How the JFS File System and Cache work.**

JFS benchmarking: **How to measure JFS performance under OS/2.**

JFS cache tuning: **How to optimize JFS performance for your job.**

What do these settings say to you?

```
[E:\]cachejfs
```

```
      SyncTime:           8 seconds
      MaxAge:             30 seconds
      BufferIdle:          6 seconds
      Cache Size:        400000 kbytes
      Min Free buffers:   8000 (    32000 K)
      Max Free buffers:  16000 (    64000 K)
```

```
Lazy Write is enabled
```

Do you have a feeling for this? Do you understand the dynamic cache behaviour of the JFS cache? Or do you just rely on the “proven” cachejfs settings that the eCS installation presented to you? Do you realise that the JFS cache behaviour may be optimized for your jobs?

Where was Journaled File System (JFS) designed for?

1986 Advanced Interactive eXecutive (**AIX**) v.1 based on UNIX System V. for IBM's RT/PC.

1990 JFS1 on AIX was introduced with **AIX version 3.1** for the **RS/6000 workstations and servers using 32-bit and later 64-bit IBM POWER or PowerPC RISC CPUs.**

1994 JFS1 was adapted for **SMP servers** (AIX 4) with more CPU power, many hard disks and plenty of RAM for cache and buffers.

1995-2000 JFS(2) (revised AIX independent version in c) was ported to OS/2 4.5 (1999) and Linux (2000) and also was the base code of the current JFS2 on AIX branch.

Just for timeline comparison:

1989 High Performance File System (HPFS) was released with 16 bits OS/2 version 1.2. HPFS supported partitions up to 64 GiB and file sizes up to 2 MiB. The maximal HPFS cache was 2 MiB, but Microsofts HPFS386 cache supported "all available memory".

1990 Microsoft Windows 3.0 used the 8.3 **File Allocation Table (FAT)** file system.

1994 Windows NT 3.1 (-W2K!) contained a pinball.sys driver to use HPFS instead of FAT.

1995 Windows NT v. 3.51 promoted **NTFS1** user rights. Windows 9x used **vFAT** and **FAT32**.

1999 Windows NT 4 and successors used NTFS.

Two major FS features kept **AIX servers** up and running:

1. **A Logical Volume or Storage Manager** that was able to mount new partitions and extend existing parts of the file system (logical volumes) without a reboot.
2. **A Journaled File System** that can rapidly restore complex directory structures after a crash.

Note that a "**stupid**" **chkdsk/fschk on FAT, HPFS or EXT2** systematically check *all* the directories and files on the partition for lost files and errors. This can take hours on large partitions.

But suppose you lost your wallet during a walk. Would you search the whole town for it?
No, you should only redo your walk.

With JFS1 (1990) IBM applied for the first time **Transactional Database Techniques** to a **File System**. This database technology was earlier used by governments and financial institutions to store and retrieve data reliably in an unstable networked environment. How does JFS works?

- All **changes** (transactions) in the file system structure (**metadata**) are logged.
- Any file (rename, create, delete etc.) transaction is only completed when **logged**.
- After a crash the JFS **chkdsk** reads the logfile to know where to search for potential errors.
- So the last intact (committed) directory structure can often be restored from the log (**logredo**).

With JFS you may lose individual files, but loss of complete directory structures is unlikely. Even if a large JFS LVM volume uses several hard disks.

IBM ported the in c written version **JFS2 (JFS) to OS/2 Warp Server for e Business (WSeB) Warp 4.5 (1999)** because the HPFS and HPFS386 file systems had major limitations:

Gordon Letwins HPFS386 code was owned by Microsoft.

HPFS has long recovery times compared to a journaled FS like NTFS.

HPFS has no Unicode Transformation Format (UTF) naming needed for internet.

HPFS was designed for an earlier generation (smaller scaled) PC systems.

Limits*	FAT16	HPFS	HPFS386	JFS	NTFS5
Partition size	2 GiB	64 GiB	64 GiB	2 TiB	16 EiB
File Size	2 GiB	2 GiB	2 GiB	2 TiB	16 EiB
Cache size	14 MiB	2 MiB	512 MiB	800 MiB	dynamic

The **OS/2 port of JFS** introduced two new features:

A **Logical Volume Manager** to do the expanding tricks on JFS volumes.

A large **JFS buffer cache**, mimicking the Unix Virtual File System.

* Though the vendors often give theoretical limits, there also exist practical limits like limitations in other drivers, the API and virtual memory address space: So the 512 MiB for the HPFS386 cache and 800 MiB for the JFS cache are more realistic then the predicate "all physical RAM".

How do the JFS File System and Cache work?

Explaining the detailed structure of the Journaled File system is beyond the scope of this presentation.

For details see the **JFS presentations** given by **Steve Best** on the web.

I will just focus on some major **File System** and **Disk Caching Concepts** and compare JFS with FAT, HPFS and NTFS.

To sharpen your focus, I start with a QUIZ.

Why does a File System Cache speed up access to the hard disk?

- A. **Magnetic Storage** is slow compared to random access memory.
- B. **File Systems** became slower as they had more overhead.
- C. The FS cache contains **Frequently Used Metada**.
- D. The FS cache contains **Frequently Used Files**.

All answers are somehow true, but only one answer is crucial for classic hard disk caching.

- E. The FS cache minimizes **physical Disk Header Movements**.

We will investigate these questions and answers theoretically and in practice..

Magnetic Storage is slow compared to random access memory.

Once at a time magnetic storage was terribly slow. Early Unix systems used tape. DOS and even Unix ran from removable disks. So buffers became an essential part of the OS Virtual File System and/or their drivers.

But today we have fast hard disks, chipsets, CPU's and drivers that can access the disk via DMA nearly at bus speed. So a software cache is not needed to read an unfragmented file fast ahead. In fact the 4-8 MiB track cache of the hard disk controller will do this for you.

DISKIO - Fixed Disk Benchmark, Version 1.18z

(C) 1994-1998 Kai Uwe Rommel

(C) 2004 madded2

Dhrystone 2.1 C benchmark routines (C) 1988 Reinhold P. Weicker

Dhrystone benchmark for this CPU: 2164987 runs/sec

Hard disk 1: 255 sides, 19457 cylinders, 63 sectors per track = 152625 MB

Drive cache/bus transfer rate: 27550 k/sec

Data transfer rate on cylinder 0: 25707 k/sec

Data transfer rate on cylinder 19455: 25274 k/sec

CPU usage by full speed disk transfers: 39%

Average data access time: 18.3 ms

Multithreaded disk I/O (4 threads): 13248 k/sec, 27% CPU usage

Realize that DISKIO reads disk sectors *directly* via the buffers of the HD driver, bypassing the File System and caches. Reading raw data sequentially from the hard disk can be very fast.

File Systems became slower as they got more overhead.

Modern File Systems support large volumes, more and larger files and new features: Logging, redundancy, long UTF File names, EAs, user rights and other file attributes. This gives FS overhead. But they also resisted fragmentation and got faster search routines.

File System	Designed for	Cache and FS design considerations
FAT8	Single user Floppy DOS	512 bytes DOS Buffers sector caching.
FAT16	16 MiB - 2 GiB partitions	FAT table cache ($2^{16}= 64$ KiB + cluster limits).
HPFS	OS/2 multiprogramming	Compact, segmented design of FS and cache.
HPFS386	OS/2 LAN server	Local user rights , more files in LAN cache.
vFAT	Windows 95 client	Dynamic cache to support FAT, LFN and OS.
FAT32	Windows 9x/ME	Idem for FAT32x partitions and files ($2^{32}=4$ GiB).
JFS1	Multi-user AIX systems	Journalled file system for AIX LVM (64 bits).
NTFS	Multi-user NT systems	Idem to support Unix features on Windows.
JFS2	Multi-user IBM systems	64 bits JFS independent of AIX.

As File Systems scaled up, became more generic and feature rich, the memory and CPU-use needed to cache FS metadata increased fast, even with clever database techniques:

For this a 16 MiB JFS cache (1/8 of 128 MiB) makes little sense. JFS was designed for large servers. If you lack RAM, CPU time or disk space, you better use HPFS (as once FAT).

Why caching metadata is essential: Understand "How NTFS Reads a File"

Below is an example of what occurs when NTFS goes to read in the 1-cluster file `\Flintstone\Barney.txt`.

1. The volume's **boot record** is read to get the cluster address of the first cluster of the MFT.
2. The first cluster of the **MFT** is read, which is used to locate all of the pieces of the MFT.
3. MFT record 5 is read as it is predefined to be the MFT record of the root directory.
4. Data cluster 0 of the **root directory** is read in and searched for "Flintstone".
5. If "Flintstone" is not found, then at least one other data cluster of the root directory needs to be read to find it.
6. The MFT record for the "Flintstone" directory is read in.
7. Data cluster 0 of the "**Flintstone**" **directory** is read in and searched for "Barney.txt".
8. If "barney.txt" is not found, then at least one other data cluster of the "Flintstone" directory needs to be read to find it.
9. The MFT record for the "Barney.txt" file is read in.
10. Data cluster 0 of the "**Barney.txt**" file is read in.

In this worst case scenario on an unmounted volume 10 slow hard disk header movements are needed to find and read Barney.txt.

QUIZ: Why is `\Flintstone\Fred.txt` approached faster?

Master File Table (MFT): A relational database that contains information about the files and directories (inodes) of a NTFS volume. It describes file names, security identifiers, timestamps, lists of cluster numbers, indexes, file attributes like "read only", "compressed" etc.

Source: <http://files.diskeeper.com/pdf/HowNTFSreadsafile.pdf>

Cache overhead: copy all buffers to cache and cache resynchronizing

File and Directory Caching generally means first **copying** the in disk or file system **buffers** placed disk sectors or blocks to **cache memory**, from where programs can read them:

Hard Disk <1> FS or disk buffers <2 > Buffer Cache <3> Program.

So caching will always slow down the first reading of any file or directory. But as the copy from memory to memory (step 2) using is fast, you will not notice it.

If the **next I/O** operation yields a **cache hit**, the slower steps 1 and 2 are avoided. A segmented LRU cache will only need to update its internal book keeping in terms of cache hits and misses.

If there is a **cache miss**, the needed disk sector must be brought into the buffer cache. If there are enough **Free Buffers**, the needed buffer is loaded in fast. As it was read in uncached.

But if there are **many cache misses (COPY A B)** the Free Buffers pool in the cache will be used up fast: If A is read in fast with 10 MB/s, free buffers diminish with 20 MB/s (A+B).

A default JFS cache uses 12,5% (1/8) of physical memory up to 64 MiB.

Default MIN and MAX values for Free Buffers are 0,5% and 1% of CacheSize.

With /Cachesize:200000 and 4 KiB blocks, Free Buffers occupy between 4 and 8 MB.

So after every $(8-4)/20 = 0,2$ seconds Free Buffers become depleted, forcing the 200 MB cache to synchronize to free 4 MiB, as long as the 20 MB/s COPY A B job runs.

The **cache overhead** thus consists of sorting 20000 pointers 5 times a second and to create enough free buffers to copy A+B to.

Cache overhead solutions

File and Directory Caching, means first **copying** the requested sectors or blocks of the File System to free buffers of the **buffer cache**, where programs can read and write to them:

File System <1> Buffer Cache <2> Program IO.

Use of buffers is inevitable, but when the cache runs out of free buffers, the cache must decide which buffers can be discarded and which buffers must be lazy written to disk.

If it does not do so, new IO (loading files, writing to disk) can only be done in the not occupied buffers.

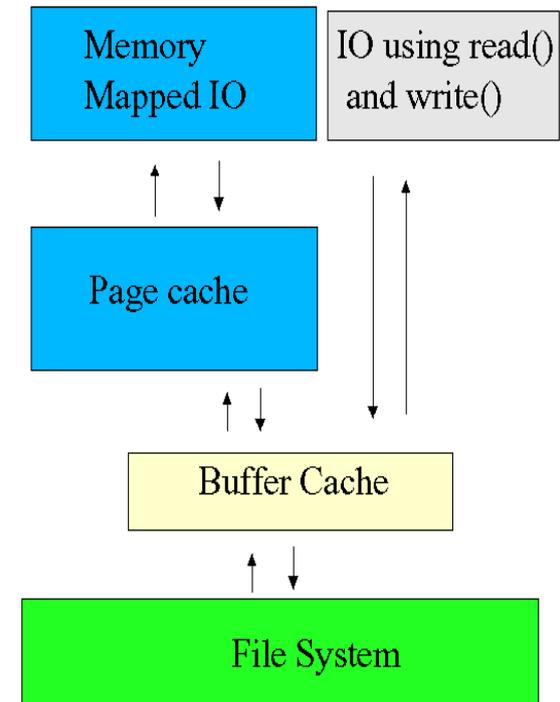
In Linux and Windows NT/XP, the newly read in or written to **4 KiB buffers** will be copied to and **stored in virtual memory**, where they are handled as pages using fast virtual memory techniques (**unified virtual memory for files and proceses**).

Buffer Cache <3> Page Cache <4> Memory Mapped IO

This **double caching** costs more CPU time, more IO cycles and more RAM as two caches (buffer and page cache) must be kept synchronized.

If there is not enough RAM, a cached file in the Page Cache could be even be swapped to disk! But it probably loads faster from its disk mirror in the page cache, than from the file system. For this reason the WPS keeps its DLLs also in virtual memory (memory-mapped file).

IO with separate buffer and page caches (unified virtual memory).

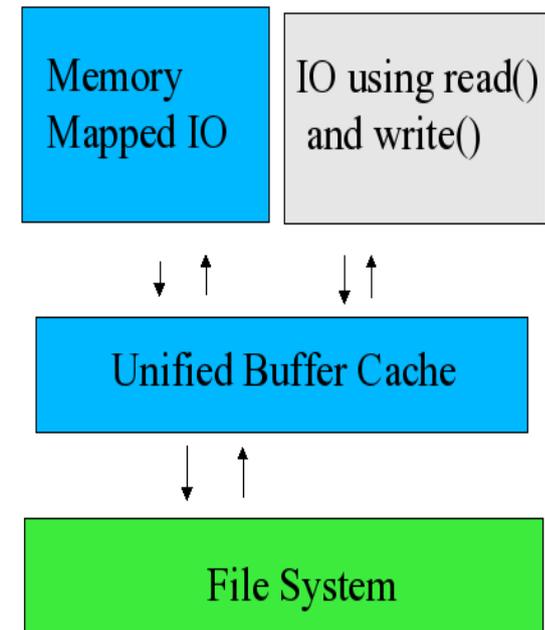


To prevent double caching, some versions of Unix and Linux do their IO via an **unified buffer cache**, where both mapping of IO in virtual memory and the read() and write() system calls use the same page cache.

Under OS/2, the memory of IFS driver caches are placed in kernel memory, and are also mapped in the 4 GiB virtual address space that processes see.

But JFS cache memory is not swapped.

IO with a unified buffer cache



The goal of any File System and Cache is to store and retrieve files fast.

Reading unfragmented data from a fast hard disk is not the problem.

The difficulty is to locate the needed disk sectors fast.

And ideally to schedule IO operations to minimize disk header movements.

A File System works best when it caches:

1. **Recently used metadata**, to find the disk sectors of files and directories fast.

2. Recently used **Files and Directories**, when they are repeatedly read or written to.

Lazy write caching, implying changes in both files and metadata, becomes very efficient when the delayed writing (**cache synchronization**) is done in a way that minimizes disk header movements:

A postman first sorts the letters, before doing his round.

Just as a well written hard disk driver has knowledge about the disk internals (**reorder the I/O queue to do R/W access faster**), a well written **FS driver and its cache** should have some knowledge of the setup and drawbacks of the File System.

But **cache synchronization algorithms** (typically LRU) are not concerned with FS layout, but only with **time stamps**. Nevertheless, as the cache sequentially reads in the needed parts of a file, the last accessed buffers of one file will have sequential time stamps.

Moreover, the **File System layout** can be designed to reduce disk header movements. It can even be optimized for specific disk cache sizes and layouts. Under HPFS and JFS this is the case, but not under FAT.

FAT16 caching

The **File Allocation Table** File System originally supported diskettes running BASIC. FAT12 formatted **diskettes** were cached via 512 bytes sector Buffers (BUFFERS=nn). When used on hard disks, a lazy write **disk cache** kept the central 16-bit **file allocation table (FAT)** and most used **directory entries** of up to 32 MiB FAT partitions in memory. DOS 4 introduced **disks clusters** spanning **2-64 disk sectors** allowing for up to **2 GiB** (64K*cluster size) FAT16 partitions, but also much slack.

Slack space: A 2 KiB file or directory entrance cannot be efficiently laid down on 32 KiB FAT16 disk cluster: 30 KiB will be unused. The 32 bit **FAT32** Table reduced the clustersize to 4 KiB, but the FAT table grew immense (2 MiB on a 2 GiB partition).

The **FAT16 table** contained the filename, extension, attribute, time stamps, size and first cluster of a file. As its records were **unordered**, it could not be searched fast unless fully cached (128 KiB).

Larger files were likely to become **fragmented**: needing more disk header movements to be read. And a **DIR** had to count all the records to measure free disk space.

Super FAT

OS/2 2.0 approached FAT16 via a kernel driver (other FS are installable).

The 32 bit hard disk **DISKCACHE** contained the 128 KiB FAT Table(s) and most used directory entrances. Its size is 48-14400 KiB.

OS/2 **Long File Names** (LFN) were supported as Extended Attributes in the mostly fragmented file **EA_DATA . SF**.

They existed independent from the LFN of vFAT used by the 32 bit Windows 9x FAT disk drivers.

Note that only 1-100 512 bytes **BUFFERS** can be placed between the disk driver and the DiskCache. This is a bottleneck.

FAT FS <1> DOS Buffers <2> FAT Cache <3> Program IO.

eCS uses BUFFERS=90, the default is only 10 (5 KiB).

High Performance File System

The 32 bit High Performance File System (HPFS) was developed for multiprogramming on hard disks.

HPFS was said to be slower than super FAT, but this was only true for small defragged hard disks (<100 MiB), on slow 386 processors with lack of RAM (HPFS needed 200 KiB + cache).

HPFS supported partitions up to 64 GiB with minimal slack space (512 bytes sectors). File sizes (incl. swapper.dat) could be up to 2 GiB with minimal fragmentation .

The central 8 MiB **seek center** (directory band) was placed strategically in the middle of the partition, minimizing disk header movements to the most accessed (meta)data. Here **sorted B(alanced) trees** enabled fast finding of metadata.

Where the fragmented FAT table behaved like a unordered staple of A4 papers, the HPFS seek center was hierarchically organized and approached like an series of indexed phonebooks, so that it could find metadata fast even when minimally cached.

File names (1-254 characters) and Extended Attributes (0-64 KiB) were laid down locally nearby the files instead of using the central FAT table and EA_DATA . SF file of super FAT.

File and directory **fragmentation** were prevented by using **free space bitmaps** in **8 MiB allocation bands** allowing for 16 Mib contiguous files.

The decentralized and redundant design (hotfixes via spare blocks) allowed for **fault tolerance** at times that hardware errors and traps were common -;)

See: <http://seds.org/~spider/spider/OS2/HPFS/hpfs.html>

The HPFS cache

The relatively small HPFS cache profited from the flat and decentralized organization of HPFS.

For this HPFS becomes exponentially faster and more efficient than FAT16 or FAT32 on > 100 MiB partitions, certainly when the FAT table cannot be cached fully.

The centrally placed 8 MiB Seek Center facilitated the looking up of the disk sectors for an previously unknown file or directory within a few disk sector reads.

Unlike the unordered FAT16 tables, that had to be fully cached (128 KiB or 256 disk sectors on a 2 GiB partition) to be usable.

As the HPFS directory paths were separately cached , the next read of a related file Fred.txt would be done at once even with a 32 KiB cache.

As less metadata were needed in cache memory, the cache could lazy write the cached file and metadata of many more applications.

The memory saved could be better be used for multitasking on systems with 8-16 MiB RAM.

Tuning the HPFS cache

IFS=C:\OS2\HPFS.IFS /CACHE:128 /CRECL:4 /AUTOCHECK:C

/CACHE:128 specifies the cache size in KiB (32-2048) and was defaulted by IBM to a tiny 128 KiB with 6 MiB RAM. Nowadays, with > 32 MiB RAM, you would set it at 2048, or if you never use HPFS, just REM the HPFS.IFS statement, freeing 200 KiB + cache RAM.

/CRECL:4 specifies the maximum cache record size (a twofold of 2-64 KiB). Setting it to large (relative to /CACHE:nn) could spoil the cache. Caching metadata is much more important!

CACHE /LAZY:ON /BUFFERIDLE:500 /MAXAGE:5000 /DISKIDLE:1000 /READAHEAD:ON

/LAZY:ON means enable Lazy Writing (recommended default: many times faster).

/BUFFERIDLE:500 buffers not used for 0,5 seconds are written to disk. The idea is that when a buffer is accessed (cache hit), it could soon (< 0,5 s) be accessed again. When this is not the case, the buffer is destined to be written to disk (if dirty) or goes to the pool of free buffers.

/MAXAGE:5000 forces frequently used buffers to be written to disk after max 5 seconds.

/DISKIDLE:1000 means that lazy write thread preferably waits until the disk is not accessed for 1 sec. During heavy IO, lack of free buffers will force acting earlier.

/READAHEAD:ON The cache uses one Read Ahead thread.

The default CACHE settings seem to work well. But they were tuned for slower systems with little RAM. For 2 MiB HPFS caches, you could increase BUFFERIDLE and MAXAGE. And with the faster seek times of modern hard disks the DISKIDLE setting can be decreased.

The Journaled File System

Like HPFS, JFS starts with the **superblock**, which contains partition information and the way JFS is organized. It has a copy that can be used by recovery tools.

Like HPFS (allocation bands), JFS is organized in repeating units called Allocation Groups.

Segmented storage prevents extreme physical disk fragmentation.

A new JFS extent can range in size from 4 KiB to 64 GiB with 4 KiB Blocks. So if you copy a 40 GiB file to a new JFS volume, it will be unfragmented.

The HPFS and JFS directory entries are sorted B trees. Sorted B trees are accessed in the way you search for a name in an library catalogue.

RAM and CPU needs of **JFS** are much higher than HPFS (4 MiB minimal, 20 MiB workable), as a 64 bits FS with UTF-16 support for much larger volumes and files has more overhead.

JFS needs some more disk space too: **System space** used by JFS on a 4 GiB data partition was 0,15% by HPFS and 0,67% for JFS with block size 512 bytes.

But note that a WPS URL object of 40 bytes uses 4 KiB with the standard block size.

Small EA are stored in the inode, or if larger in separate blocks.

Every **object** (file, directory, link) in the file system is represented by an **i(ndex)node #**.

The inode table contains information like: file type, file size, used blocks, time stamps (creation, last modified and acces), user rights (uid, gid) and the availability of links and extended attributes.

```
G:\TEMP\test>ls -ila
```

```
total 172
```

```
 9288649302917795134 drwxrwxrwx 1 0 0      0 2009-10-27 23:59 .
11038877574429749475 drwxrwxrwx 1 0 0      0 2009-10-27 23:57 ..
 2302304912057420540 drwxrwxrwx 1 0 0      0 2009-10-27 23:59 dir
12559417848390446504 -rw-rw-rw- 1 0 0 163109 2009-07-13 00:21 file.png
 3558792758389222959 -rw-rw-rw- 1 0 0      828 2009-01-14 23:44 file.txt
```

Question: what counts as Total 172 in var? What are the 1 0 0 for dirs?

```
33308 -rw-r--r--    1 sjoerd users      129153 Sep 20 2008 schermafdruck1.png
```

The file names of inodes are found in a table the parent directory. If a file is erased, its entrance in the directory is erased.

For this a lost file can only be identified by June by its inode number!

ls -ila shows: inode number, user rights, number of links to the file, User ID, Groups ID, time stamps, file name.

Setting the JFS cache size

JFS is loaded as follows:

```
DEVICE=C:\OS2\BOOT\UNICODE.SYS
```

```
IFS=C:\OS2\JFS.IFS AUTOCHECK:* /CACHE:128000 /G:0 /U:0
```

With /G and /U you set the Groups and User IDs (here root).

The JFS cache size (in KiB) can be very large. It is mapped in the system arena, so its size is only limited by:

The amount of free physical memory.

Your Virtual Address Limit.

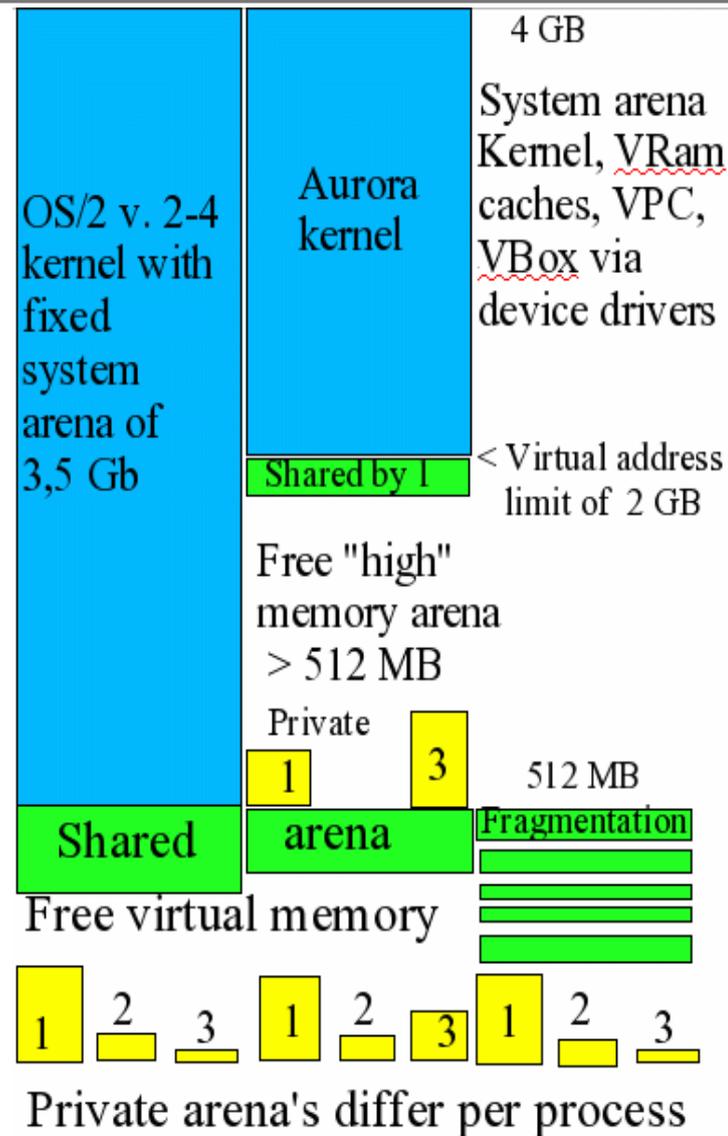
System memory already occupied (PCI, video, drivers,..)

In practice I could only load JFS caches <1 GiB in RAM.

Even after trimming unused video ram (gaoption vidmem 32). Cache sizes up to 600 MiB worked fine for me.

IBM defaulted JFS cache size to 1/8 of physical memory (max 64 MiB), so 16 MiB with 128 MIB RAM. But JFS needs at least 20 MiB to work efficiently.

As modern eCS systems will seldom have lack of physical RAM, spending unused free memory to the JFS cache seems reasonable. But of course the memory spent to the cache can not be given to VPC.



JFS cache design

The JFS cache is a segmented LRU buffer cache:

Buffer: The unit to be cached is a 4 KiB JFS buffer.

All JFS IO must go through 4 KiB JFS cache buffers.

LRU: Least Recently Used Buffers are first discarded or if dirty first written to disk by the **cache synchronization** mechanism, to create Free Buffers for still uncached IO.

Segmented: implies a kind of cache hierarchy.

The cache makes a division between data and metadata. Caching the directory and inode of a file (**metadata**) is more important than the file content itself (**data**).

The other division is that between the by at least **one cache hit** justified **protected buffers** (up to 2/3 of cache) and newbies (**probationary buffers**) that were recently read into the cache. They could be DLLs loaded once or MPEG files played once, thus spoiling the cache.

All freshly from disk read in or any newly to the disk written file is first placed in a **Free Buffer**. If this now once written to **Probationary Buffer** is accessed again, it promotes to the **Protected Buffer** segment. But without cache hit is considered as less important and stays probational.

The Least Recently Used Probational Buffers will be first thrown out of the cache to make place for the by uncached IO needed **Free Buffers**. And if more Free Buffers are needed for IO, the LRU protected buffers are used (they also degrade to the Probational segment if $slrun > slruN$).

Protected buffers: had at least one cache hit
 cbufs_protected (data)
 jbufs_protected (metadata)

$slrun$ (segmented lru number) =
 $cbufs_protected + jbufs_protected$
 Max $slrun = slruN$ (2/3 of cachesize)

Probationary buffers: had no cache hit
 cbufs_probationary (data)
 jbufs_probationary (metadata)

Free buffers: accept data (-> probationary)
 $maxfree > nfreecbufs > minfree$

```
[E:\CACHEJFS.EXE /LW:8,30,6 /MINBUFFER:16000 /MAXBUFFER:32000
```

```

    SyncTime:          8 seconds
      MaxAge:          30 seconds
    BufferIdle:         6 seconds
    Cache Size:    400000 kbytes
    Min Free buffers:  16000 (   64000 K)
    Max Free buffers:  32000 (  128000 K)

```

Lazy Write is enabled

```
[E:\]cstats
```

```

cachesize      100000      cbufs_protected      43148
hashsize       65536      cbufs_probationary   6394
nfreecbufs     47941      cbufs_inuse           0
minfree        16000      cbufs_io               0
maxfree        32000      jbufs_protected       1565
numiolru        0         jbufs_probationary    946
slrun          44713      jbufs_inuse           0
slruN          66666      jbufs_io               0
Other           6         jbufs_nohomeok        0

```

JFS utilities like `cstats.exe` can be found at <ftp://ftp.netlabs.org/pub/openjfs/>.

The eComStation 1.2 client started the JFS driver with:

JFS cache settings

IFS=C:\OS2\JFS.IFS /LW:5,20,4 AUTOCHECK:*

But the IBM Warp Server for eBusiness server default was:

IFS=C:\OS2\JFS.IFS /LW:64,256,8 AUTOCHECK:*

Both yield a (too small) **64 MiB JFS cache** on systems with 512 MiB or more memory.

[F:\]cachejfs

```

SyncTime:          5 seconds
MaxAge:            20 seconds
BufferIdle:        4 seconds
Cache Size:        65536 kbytes
Min Free buffers:  327 (    1308 K)
Max Free buffers:  655 (    2620 K)

```

Lazy Write is enabled

SyncTime is the maximal time span after which a sleeping cache must synchronize its contents. SyncTime was 64 seconds for WSEB, but was reduced to 5 seconds on eCS. Why?

MaxAge of HPFS was 5 seconds. But the larger lazy write JFS cache writes dirty buffers every SyncTime*4 (20 on eCS, 256 on WSEB) seconds to disk, if not forced to do so earlier.

The **BufferIdle** setting of 4 seconds (8 with WSEB) is much larger than the 0,5 s of the smaller HPFS cache. Probational dirty buffers not rewritten to within BufferIdle seconds must be written to disk. If the cache is forced to synchronise (by heavy IO), the only once accessed dirty buffers must after BufferIdle seconds be send to the lazy write thread. But if hit again within BufferIdle seconds, they promote to the protected part of cache that writes them after MaxAge to disk.

Changes in internal cache status, like probationary buffers becoming protected, are not implemented as memory movements. Only small entries (**pointers**) in the **cache tables** are updated. For this reason **cache synchronization overhead** can be kept small even with large caches. But never underestimate the scale of it:

A 400 000 KiB JFS cache has 100 000 entries to maintain.

[E:\]cstats

cache size	100000	cbufs_protected	21863
hash size	65536	cbufs_probationary	10447
nfreesbufs	64710	cbufs_inuse	0
minfree	8000	cbufs_io	0
maxfree	32000	jbufs_protected	1453
numiolru	0	jbufs_probationary	1521
slrun	23316	jbufs_inuse	0
slruN	66666	jbufs_io	0
Other	6	jbufs_nohomeok	0

Here I have mostly (64710) Free Buffers, but if the cache becomes saturated (slrun=slruN), 66666 Protected and quite a lot of Probationary buffers have to be maintained.

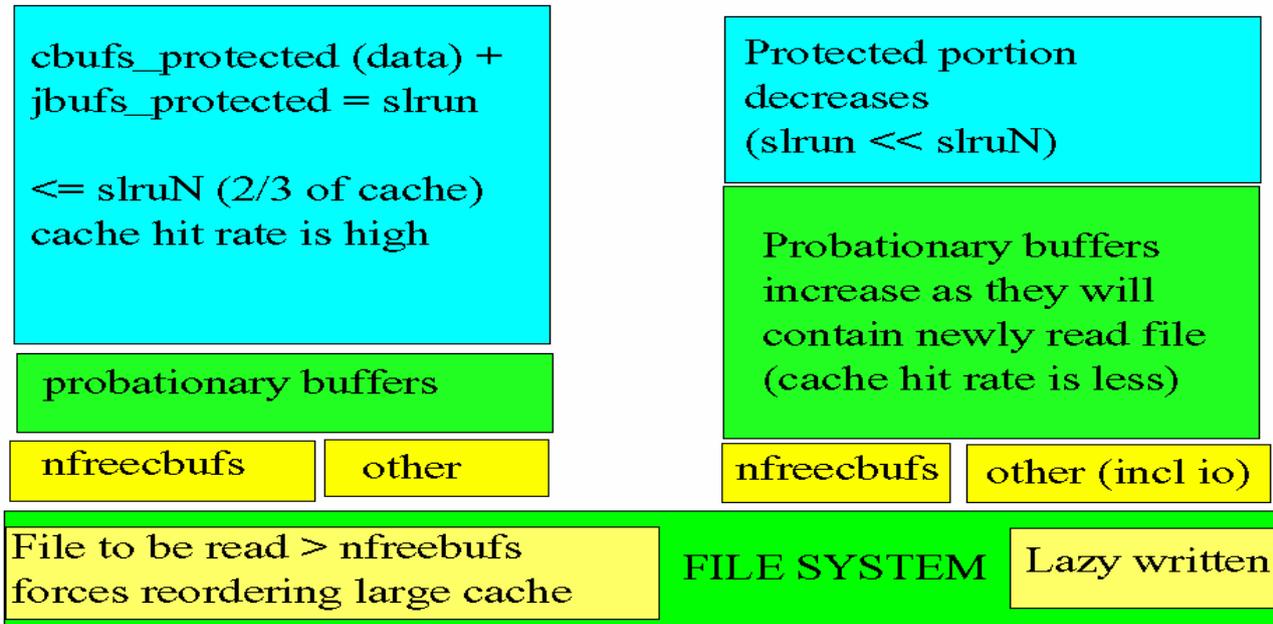
And when a saturated cache must suddenly read in big files, IO may increase extra, when the cache is forced to flush dirty buffers that have not been updated (via MaxWait).

The clusters of a newly to be read or written to the JFS cache file are first placed in **free cache buffers** (freecbufs).

The written to free buffers become **probationary buffers**.

If read again or written to (**cache hit**), probationary buffers promote to the **protected segment**. This most static part of the cache may occupy up to slruN clean and dirty buffers.

Saturated LW cache => Heavily reorganizing cache.



To get enough free buffers to cache large files compared to cachesize, the cache has to synchronize. The easy to discard least recently used (LRU) probationary buffers are processed first, eventually followed by the LRU protected buffers, until $minfree < nfreecbufs$.

If cached IO is high (backups, io benchmarks), massive lazy writing may be forced. Many dirty buffers in the protected segment must be written to disk (via io_buffers). But a saturated cache will contain dirty buffers of many files scattered around the disk. So the lazy write thread can take a long time as it cannot write the dirty buffers sequentially. Though asynchronous randomly file IO via the lazy writing cache is much faster than synchronous (first in the IO queue, first served) writing without a cache used by databases, swapper.dat, ini files, the scale of it imposes a problem, that may result in a temporally freezing of the system with a large cache.

Some JFS performance issues

Speed matters, but **Data Integrity** and **System Stability** come always first.

No JFS related Trap Errors should happen.

Cache actions should be unnoticeable to the user.

Use of the lazy write cache should not give more risk of data loss.

Old data should be recovered completely after a crash.

Speed issues

In benchmarks JFS proved superior compared to JFS to HPFS and HPFS386.

But note that I used rather large JFS caches (32-800 MB). As JFS was not designed for small systems.

No JFS related Trap Errors should happen.

The **JFS driver code** is executed in **kernel space**. So if something goes unexpectedly wrong, you will see a **trap error**.

Most of the trap errors seem to have been resolved by IBM with fixes for OS/2 Warp server.

Unexpected effects brought about by new ACPI or other hardware drivers could destabilize JFS.

As the OS/2 kernel is not any more maintained, fixing kernel related problems is difficult.

Unexpected changes in the JFS File System, when sharing JFS with Linux are even more risky.

It is likely that **JFS on Linux code** will divert from the original JFS OS/2 code, as the needs of niche player OS/2 are neglected.

JFS will be optimized for other operating systems.

JFS on OS/2 and Linux will become different species.

Also note that a **Linux LVM** does not act as the OS/2 LVM.

Trap errors and data loss are likely, when JFS is approached by different drivers in an inconsistent way.

Cache actions should be unnoticeable to the user.

In normal OS/2 practice JFS caching works transparently to the user.

But if you run Sysbench of Trevor Hemsley, you may run into trouble.

When I ran the random Cached disk read and write of Sysbench on JFS on a single core processor, I noticed **PM and WPS Desktop freezing** for seconds to minutes depending on the test and on cache size.

The Desktop was frozen. Even Watchcat and CAD handler did not respond.

After a period, OS/2 live revived as before. The Watchcat and CAD handler pop up menu's did appear, but time critical operations network operations could be lost.

On my dual core laptop with SMP kernel this disk activity related PM and WPS freezing did not occur. One core seemingly "used" 99,9%, but the other kept responsive.

QUIZ: What happened? CLUES are found in the WSEB APARs:
 NETBENCH TEST ON JFS CAUSES A HANG DUE TO INSUFFICIENT FREE
 CACHE BUFFERS. LARGE XCOPY ON A JFS DRIVE LEADS TO A HANG.
 JFS HANGS WHEN DOING I/O ON THE SERVER FROM A LARGE NUMBER
 OF CLIENTS

Sadly IBM has removed the entries. But hangs were associated with sudden heavy IO through the cache.



What happens during JFS cache related freezing?

JFS must run a synchronizing kernel thread when it runs out of free buffers.

It is also forced to write dirty buffers to disk.

When a single core CPU is in kernel modus, pre-emptive multitasking is postponed.

The synchronizing thread can take minutes, when a large JFS cache (>200 MIB) is "spoiled" by an impossible to be cached (efficiently) disk task. Or when large amounts of randomly written data have to be lazily written to disk. So a single core system freezes, until the kernel has done its necessary JFS job.

Please do not reset the system before that or data loss will occur!

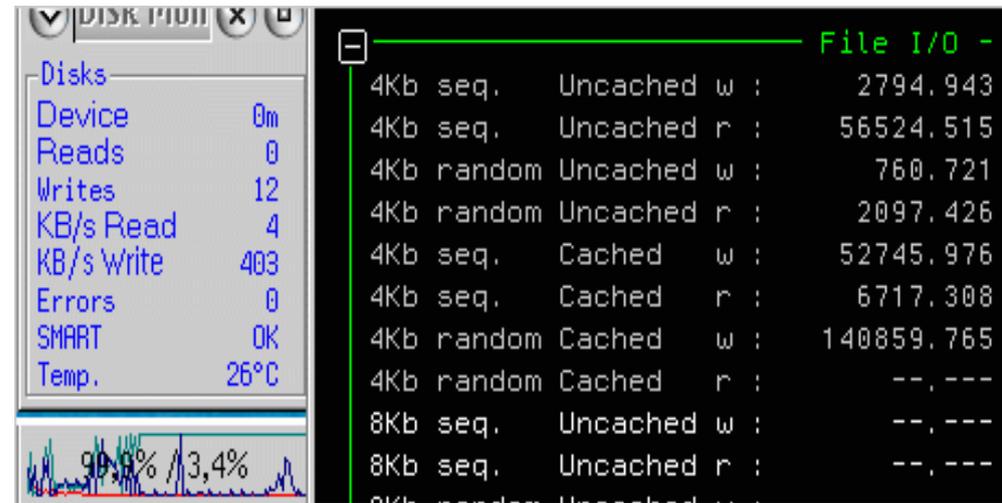
With large caches JFS works much better on SMP systems. One core can be occupied by JFS kernel threads, whilst the other core can serve PM and the WPS (see picture). But for OS/2 WSEB or OS/2 4.5 single core systems "remedies" for cache related system lockups were:

Decreasing the cache size (64 Mib became the default max)

Decreasing the scheduled synchronization times.

Increasing free buffers (not mentioned).

A smaller Cache has a shorter synchronization time, but increasing the free buffer portion of the JFS cache before doing heavy IO seems more reasonable.



Use of the lazy write cache should not give more risk of data loss.

Lazy writing dirty buffers to disk (an asynchronous write) speeds up disk access tremendously:

- There are less disk writes to files and directories.
- Writing to disk occurs in efficient bursts (as with SCSI).

Both factors reduce the chance that the computer is powered off during a write to essential FS structures.

Of course, the disk buffers must first be flushed before shutting down the PC. Without CAD or shutdown you will lose data. And probably more on larger caches with long synchronisation times.

I certainly lost some data because of bad RAM chips, CPU overheating, empty laptop batteries and trap errors.

But thanks to metadata **logging** I seldom lost complete directories with JFS caches of 400 MB.

The picture taken of a lost \OS2 DIR was on drive F using a 32 MiB HPFS386 cache (and bad memory).

So I always keep dsync backups (on JFS of course)

File Name	Size	Date/Time
..		
UP-DIR		04-11-09 21:36
SUBDIR		19-12-05 23:51
SUBDIR		13-01-06 16:28
SUBDIR		31-03-07 20:58
SUBDIR		20-09-07 1:10
SUBDIR		02-09-08 23:28
I0233474.RCN	5573	12-11-03 19:17
I0233475.RCN	6665	12-11-03 19:17
I0233476.RCN	5864	12-11-03 19:17
I0233477.RCN	5912	12-11-03 19:17
I0233478.RCN	5913	12-11-03 19:17
I0233479.RCN	735	12-11-03 19:17
I0233480.RCN	20084	12-11-03 19:17
I0233481.RCN	878	12-11-03 19:17
I0233482.RCN	3985	12-11-03 19:17
I0233483.RCN	36	12-11-03 19:17
I0233484.RCN	18106	12-11-03 19:17
I0233485.RCN	18082	12-11-03 19:17
I0233488.RCN	50947	29-10-09 13:29
I0233489.RCN	33998	29-10-09 13:29
I0233490.RCN	41697	29-10-09 13:29
I0233491.RCN	46196	29-10-09 13:29
I0233492.RCN	51500	29-10-09 13:29
I0233493.RCN	87283	29-10-09 13:29
I0233494.RCN	74850	29-10-09 13:29
I0233495.RCN	38512	29-10-09 13:29
I0233496.RCN	223	29-10-09 13:29
I0233497.RCN	223	29-10-09 13:29
I0233498.RCN	2909	29-10-09 13:29
I0233499.RCN	6945	29-10-09 13:29
I0233500.RCN	37426	29-10-09 13:29
I0233501.RCN	119653	29-10-09 13:29
I0233502.RCN	144918	29-10-09 13:29
I0233503.RCN	362403	29-10-09 13:29

How to do optimize JFS performance for your job.

```
[E:]CACHEJFS.EXE /LW:8,30,6 /MINBUFFER:8000 /MAXBUFFER:32000
```

SyncTime: 8 seconds

MaxAge: 30 seconds

BufferIdle: 6 seconds

Cache Size: 400000 kbytes

Min Free buffers: 8000 (32000 K)

Max Free buffers: 32000 (128000 K)

Lazy Write is enabled

Say you want to do heavy IO: benchmarking, backup, etc.

- **Would you reboot to decrease the cache size to a safe 64 MiB?**
- **Would you shorten the sync times?**
- **Would you disable lazy write?**

No increasing, Min Free Buffers to 16000 (64 MB) and Max Free Buffers to 84000 (336 MB MB), reduces the to be synchronized cache portion to 64 MB, but leaves you with plenty of useful IO buffers. See: cstats under sysbench.

```
[E:\]CACHEJFS.EXE /LW:8,30,6 /MINBUFFER:16000 /MAXBUFFER:84000
```

```

      SyncTime:          8 seconds
      MaxAge:            30 seconds
      BufferIdle:         6 seconds
      Cache Size:       400000 kbytes
      Min Free buffers:  16000 (   64000 K)
      Max Free buffers:  84000 (  336000 K)

```

Lazy Write is enabled

```
[E:\]cstats
```

```

cachesize      100000      cbufs_protected      16003
hashsize       65536      cbufs_probationary    10
nfreecbufs     53545      cbufs_inuse           0
minfree        16000      cbufs_io              30396
maxfree        84000      jbufs_protected       21
numiolru       30396      jbufs_probationary    2
slrun          16024      jbufs_inuse           0
slruN          66666      jbufs_io              0
Other          6         jbufs_nohomeok        17

```

Cache Stats logging

Say you have a File or Web Server then may want to know how the cache is used.

Cstats.exe can tell you at any time how the Cache Buffers are being used.

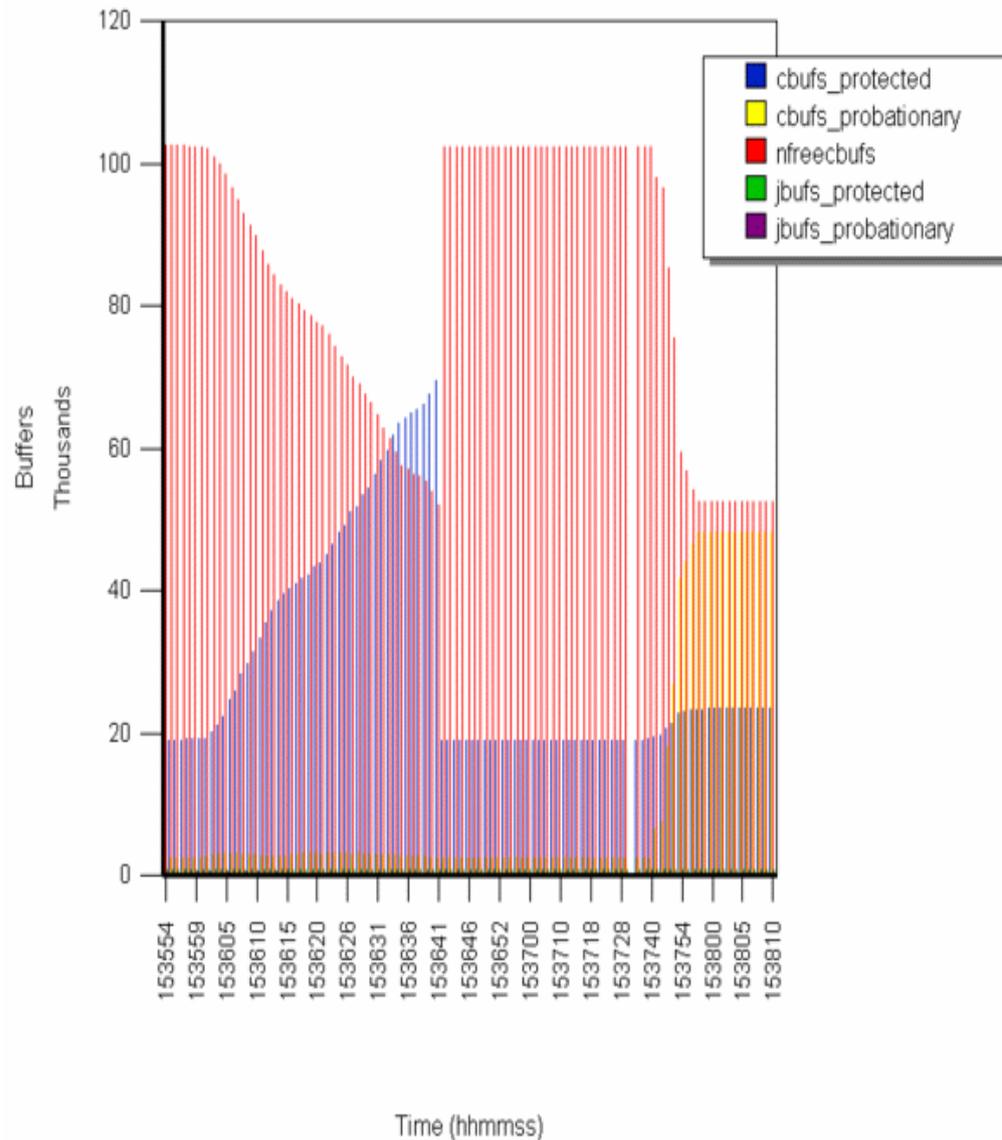
It does not tell you the amount of cache hits, but regular samples show you the dynamics of the cache. So you can tune Cache RAM and BUFFERS.

To ease analysis I use REXX scripts that converts the cstats output to csv-format.

The output can be imported in a spreadsheet or database.

See: www.sjoerd-visser.demon.nl/ecs-s2/jfs.html

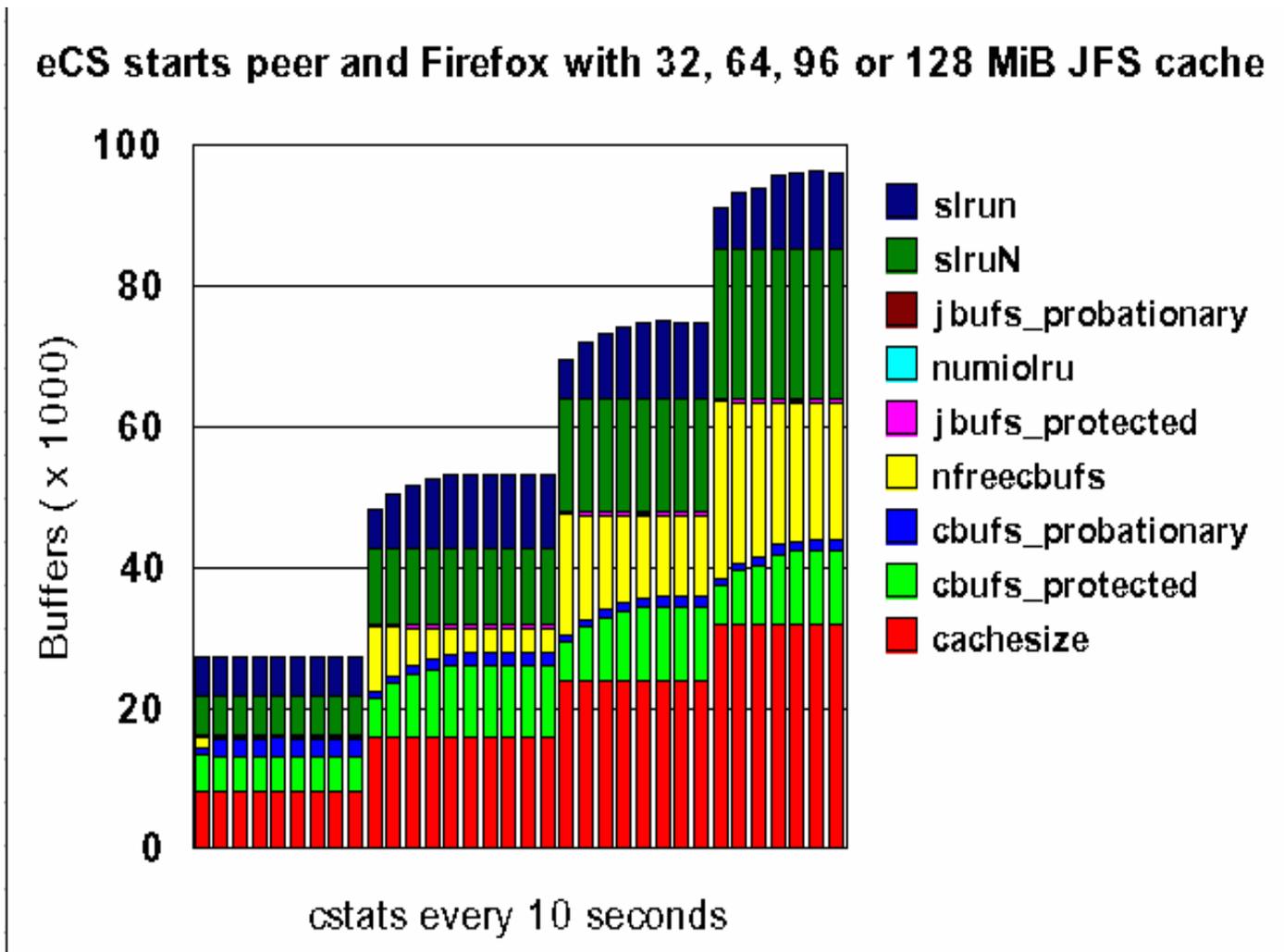
Loading and saving a VPC guest (385 MiB)

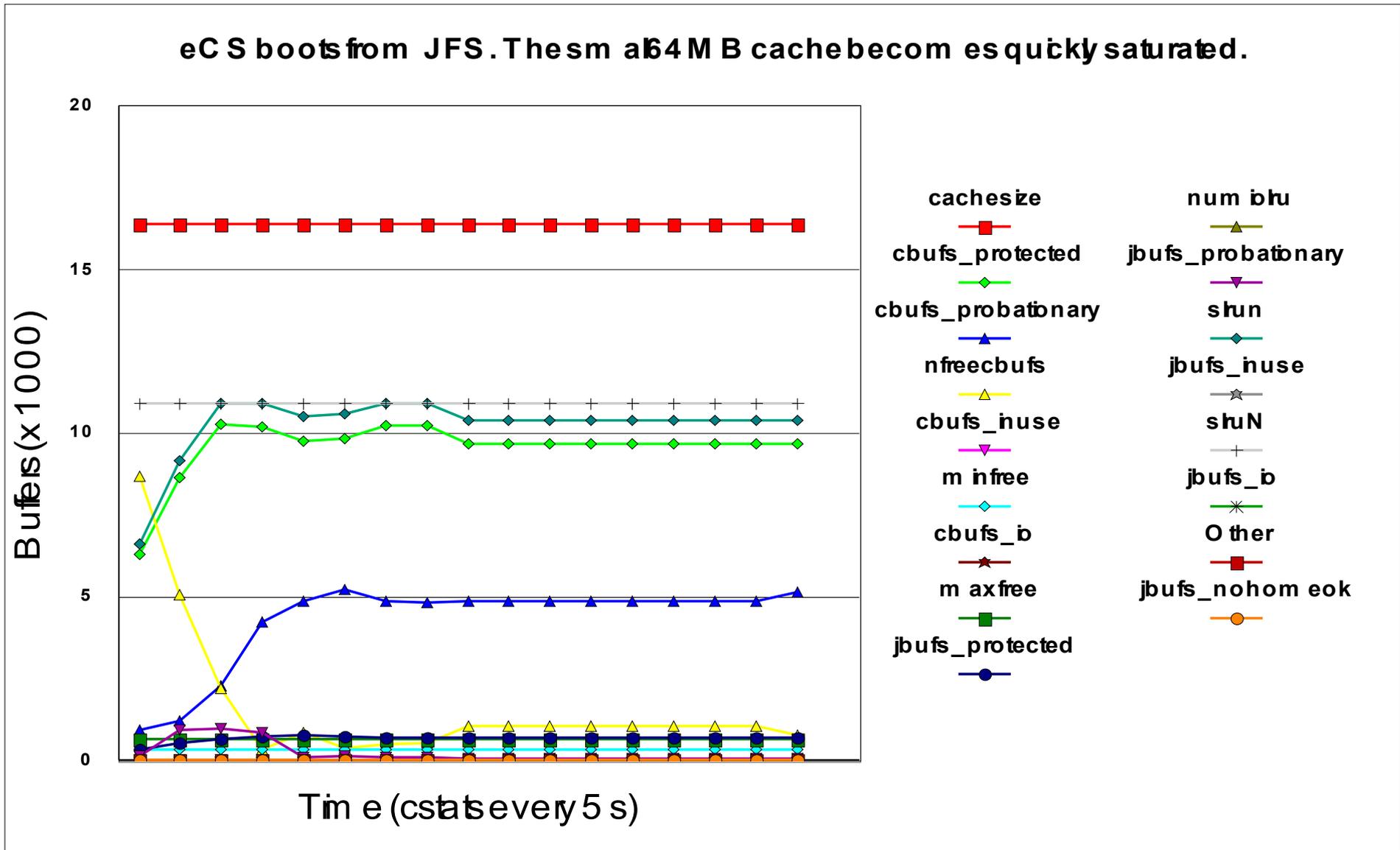


A small 32 MB JFS cache soon runs out of free buffers (yellow).

Larger JFS caches start with more free buffers and have more space to keep protected buffers that contribute most to cache efficiency.

Note: the amount of protected buffers (slrun) can grow up to $slruN = 2/3$ of cachesize.





During cache initialisation (T0) all buffers are still free (nfreebufs = cachesize).