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 do 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
```

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 **NTFS**1 user rights. Windows 9x used **FAT32**.

1999 Windows 2000 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** will 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 wil just redo your walk.

With JFS1 (1990) IBM applied **Transactional Database Techniques** to a **File System**. Technology that was used by governments and financial institutions to store data in large networked environments.

- 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.

IBM ported the in c written version **JFS2 (JFS) to OS/2 Warp Server for e Busines** (**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 OS/2 practice..

Magnetic Storage is slow compared to random access memory.

Magnetic storage is relatively slow. Early Unix systems used tape. DOS ran from removable disks. To prevent slow disk header movements, buffers became an essential part of the OS.

But current hard disks can access the disk via **DMA** nearly at **bus speed**. So a software cache is not needed to read a file fast ahead. In fact the 4-8 MiB internal 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**

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

File Systems became slower as they had 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.

FS	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**	**Journaled** 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 scale up, become more generic and feature rich, the memory needed to cache FS metadata increases fast, even when using 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 stick to HPFS.

Why caching metadata is essential: "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: ???

Cache overhead

File and Directory Caching, means first **copying** the in disk driver **buffers** placed disk sectors or blocks to **buffer cache memory**, before programs can read them:

Hard Disk <1> Disk Buffers <2 > Buffer Cache <3> FS driver <4> Program.

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

- If the **next I/O** operation yields a **cache hit**, steps 1 and 2 are avoided. The cache will only need to reorder its internal book keeping.
- It there is a **cache miss**, the needed disk sectors must be brought in the buffer cache.
- If there are enough **Free Buffers**, the needed sectors are loaded in fast.
- 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 0,25 and 0,5 % of CacheSize.

So with /Cachesize:200000 Free Buffers are between 500 and 1000 KiB.

So after every 500/20 ms (25 ms) Free Buffers become depleted, forcing the 200 MB cache to synchronize to free 500 KiB, as long as the 20 MB/s COPY A B job runs.

This **cache overhead** (sorting 50000 entries to 4 KiB buffers every 25 ms) can be a reason to increase Free Buffers when using a large cache during heavy IO.

Cache overhead

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

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

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

Buffer Cache <3> Page Cache

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

If their is not enough less RAM a file could be cached in swapspace!!

Cache overhead

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

This prevents double buffering.

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

- Reading unfragmented files 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.

Bit **cache synchronization algorithms** (typically LRU) are only concerned with **time stamps.** Nevertheless, as the sequentially read in clusters of a file will have similar time stamps, they are also more likely to be flushed to disk if .

The **File System layout** can be designed to reduce disk header movements. It can even be designed to serve the stupid disk cache algorithms.Under HPFS and JFS this is te cae,

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, **Disk caches** 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 (not a installable FS).

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

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

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

Note that only 1-100 512 bytes **BUFFERS** are placed between the disk driver and the DiskCache. This is a bottle neck.

eCS uses BUFFERS=90, the default is 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 300 KiB + cache).

HPFS supported partition sizes of 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 mess of unordered A4 staples, the HPFS seek center was organized and approached like an indexed phonebook, that found metadata fast even if minimally cached.

File names (1-254 characters) and Extended Atributes (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 used for multitasking.

Setting HPFS.IFS and its 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.

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

/LAZY:ON means enable Lazy Writing (always recommended: 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.

Note that **Cached Metadata** had a protected status compared to file data. In practice the default CACHE settings work well.

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 **extents**.

An extent behaves like a logical unit, managing the resources of the file system locally.

Files in the same directory will be preferably be placed in the same extent.

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 directory entries are sorted B trees.

Sorted B trees are accessed in the way you search for a name in a phonebook, or look for page 150 in a book.

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 blocksize 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 seperate 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
 928864930291779513 drwxrwxrwx 1 0 0      0 2009-10-27 23:59 .
1103887757442974947 drwxrwxrwx 1 0 0      0 2009-10-27 23:57 ..
 230230491205742054 drwxrwxrwx 1 0 0      0 2009-10-27 23:59 dir
1255941784839044650 -rw-rw-rw- 1 0 0 163109 2009-07-13 00:21 file.png
 355879275838922295 -rw-rw-rw- 1 0 0    828 2009-01-14 23:44 file.txt
G:\TEMP\test>ls -ila
total 12
 928864930291779513 drwxrwxrwx 1 0 0   0 2009-10-28 00:04 .
1103887757442974947 drwxrwxrwx 1 0 0   0 2009-10-27 23:57 ..
 230230491205742054 drwxrwxrwx 1 0 0   0 2009-10-27 23:59 dir
 355879275838922295 -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 schermafdruk1.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!

Setting the JFS cache size

JFS is loaded as follows:

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

IFS=C:\OS2\JFS.IFS AUTOCHECK:* /CACHE:128000

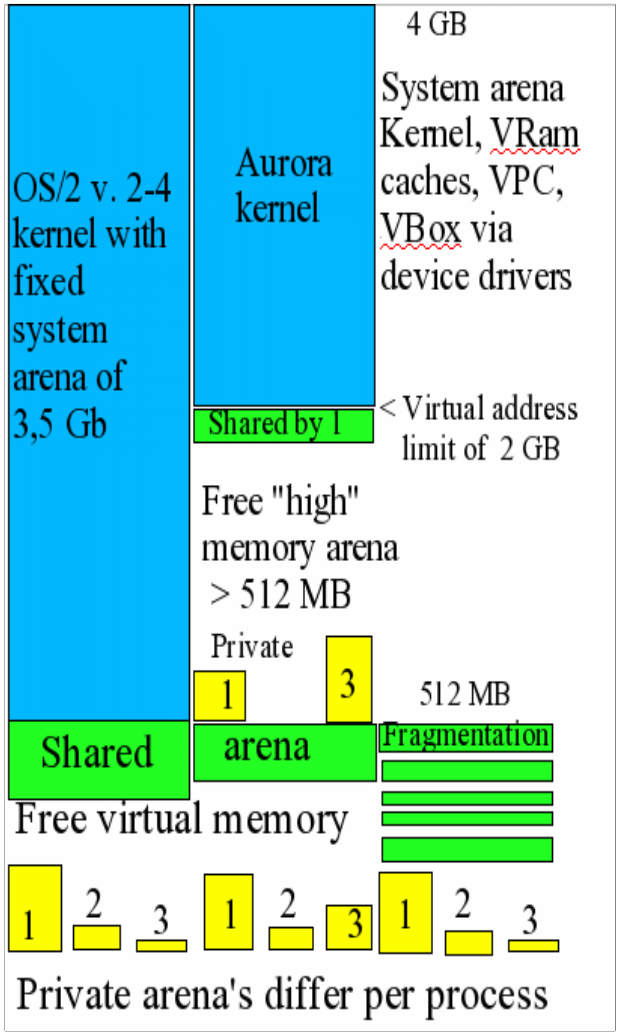
The JFS cache size (in KiB) can be very large.

It is is placed and executed in kernel space, so its size is only limited by:

- The amount of 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.

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.

LRU is the **FIFO**: First in, First out cache algorithm.

Segmented: implies a kind of **cache hierarchy** .

Just stupidly applying FIFO rules to all cache buffers would spoil the cache and reduce its effectiveness.

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

The simplest 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).

Setting the JFS cache

```
[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
```

eComStation 1.2 started the JFS driver with:

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

The IBM Warp Server for eBusiness 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 interval at which the cache synchronizes its contents. It was 64 seconds for the WSEB File Server, but was reduced to 5 seconds on eCS. Why?

Maxage of HPFS was 5 seconds. The larger JFS cache writes all dirty buffers every 20 (eCS) or every 256 seconds (WSEB) 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 seconds of the small HPFS cache. But with more Free buffers (1308 KiB) the larger 64 MiB cache, JFS can wait 4 (or 8) seconds to see if a probationary buffer can be promoted to the protected status (if accessed) or must be discarded as unused. .

Of course, **changes in cache** status of 4 KiB Buffers (probationary, protected) are not implemented as memory movements as my diagram suggests.

Only their small entries (**pointers**) in the **cache tables** are updated. So that the cache synchronization and lazy write threads know which buffers to deal with.

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
cachesize     100000   cbufs_protected       21863
hashsize       65536   cbufs_probationary    10447
nfreecbufs     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.

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 comparing 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 several 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.

What happens during JFS cache related freezing?

JFS must run a synchronizing kernel thread when it runs out of free buffers. It is then forced to write dirty buffers to the File System. The problem is that Sysbench and other utilities can create them faster in memory, than the cache can write them 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.

So the system freezes, until the kernel has done its necessary JFS job.

- Please do nor reset the system before that or data loss will occur!

With large caches JFS works much better on a SMP system.

But for OS/2 WSEB 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 heavy IO seems more reasonable.

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

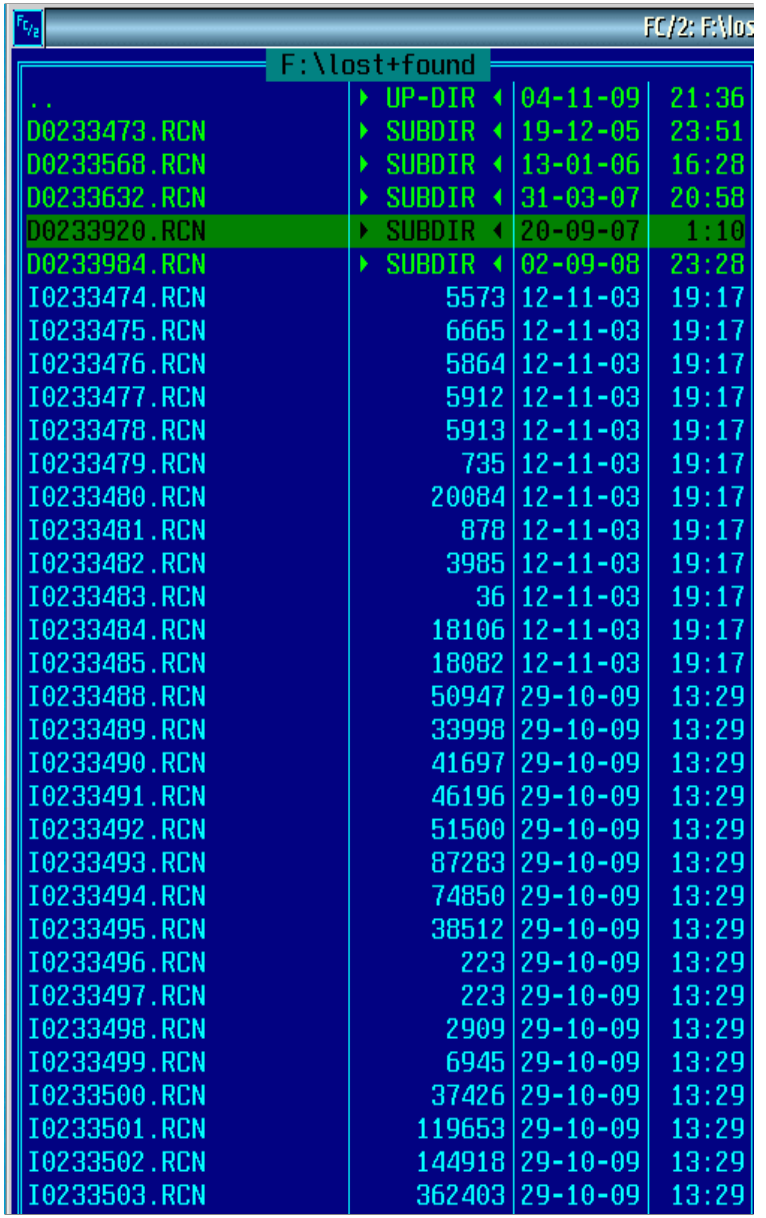
Of course, the disk buffers must first be flushed before shutting down the PC.

And I certainly lost some data because of bad RAM chips, 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)



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?

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                  0
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
```

CacheStat logging

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

Cstats tells You How Cache Buffers being are used.

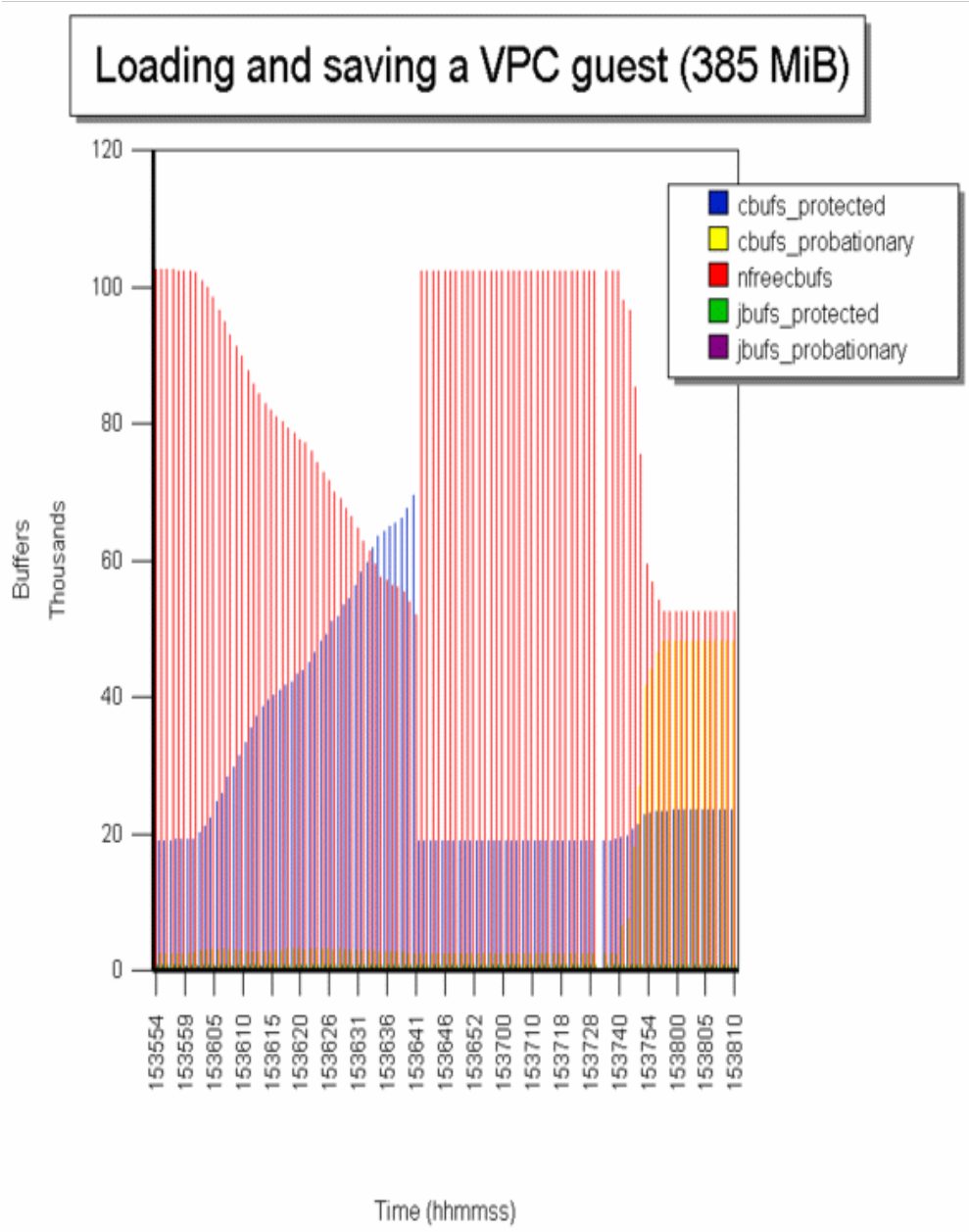
It does not tell you the amount of cache hits, but it will tell you how your RAM is used.

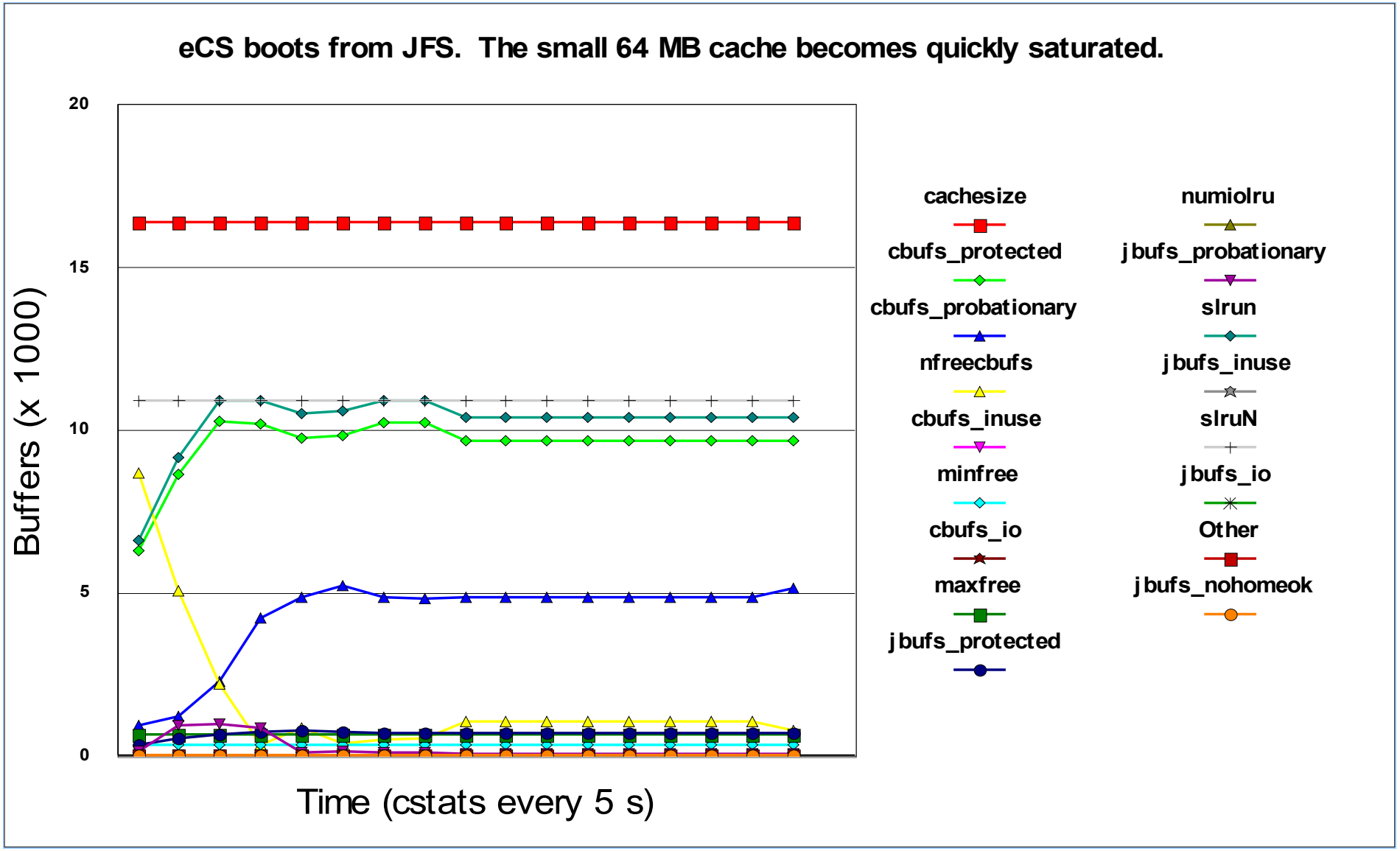
So you can tune Cache RAM and BUFFERS.

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

The output can be imported in a spreadsheet or database.

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

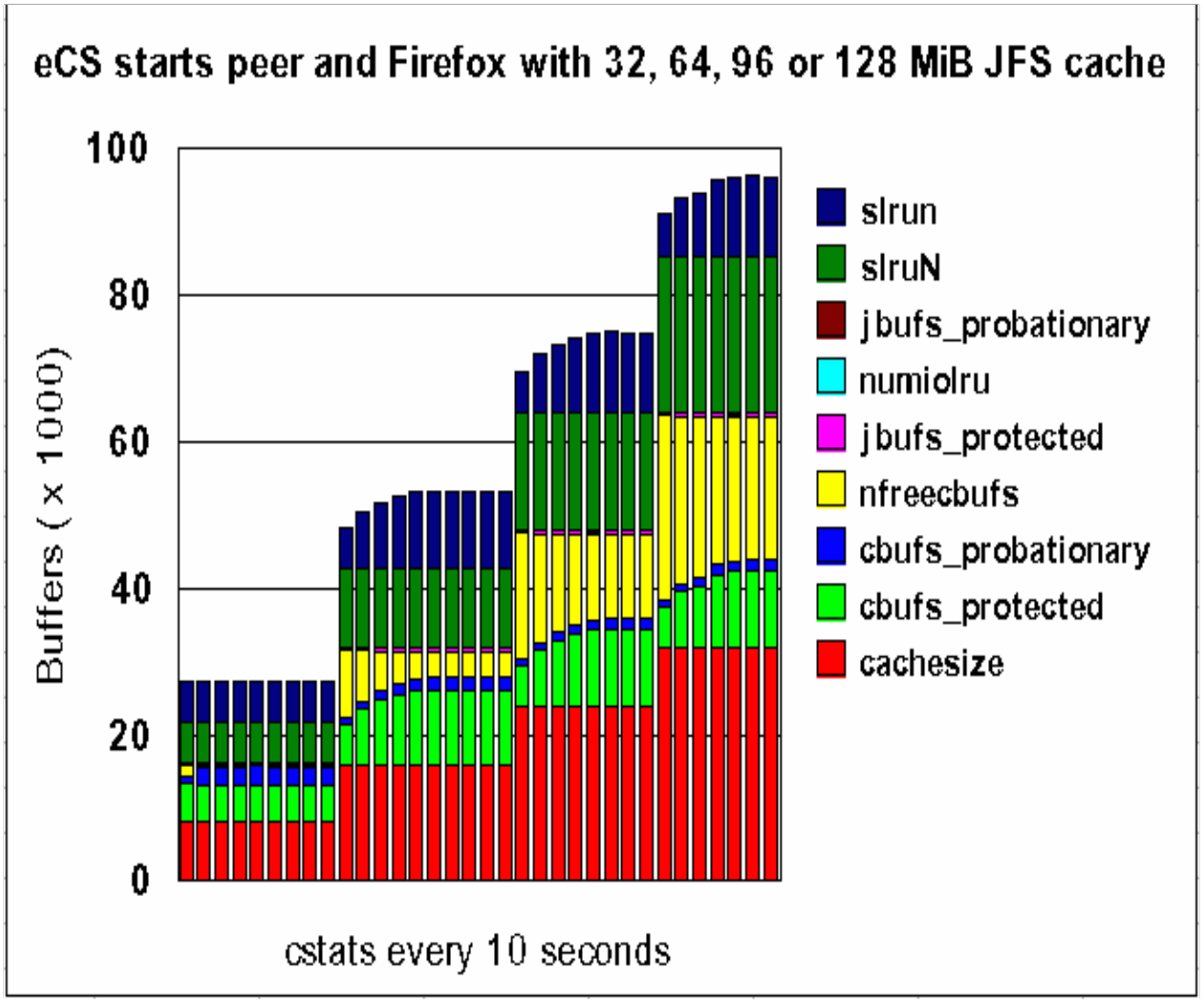



eCS boots from JFS. The small 64 MB cache becomes quickly saturated.
Buffers (x 1000)
20
15
10
5
0
Time (cstats every 5 s)
cachesize
cbufs_protected
cbufs_probationary
nfreecbufs
cbufs_inuse
minfree
cbufs_io
maxfree
jbufs_protected
numiolru
jbufs_probationary
slrun
jbufs_inuse
slruN
jbufs_io
Other
jbufs_nohomeok

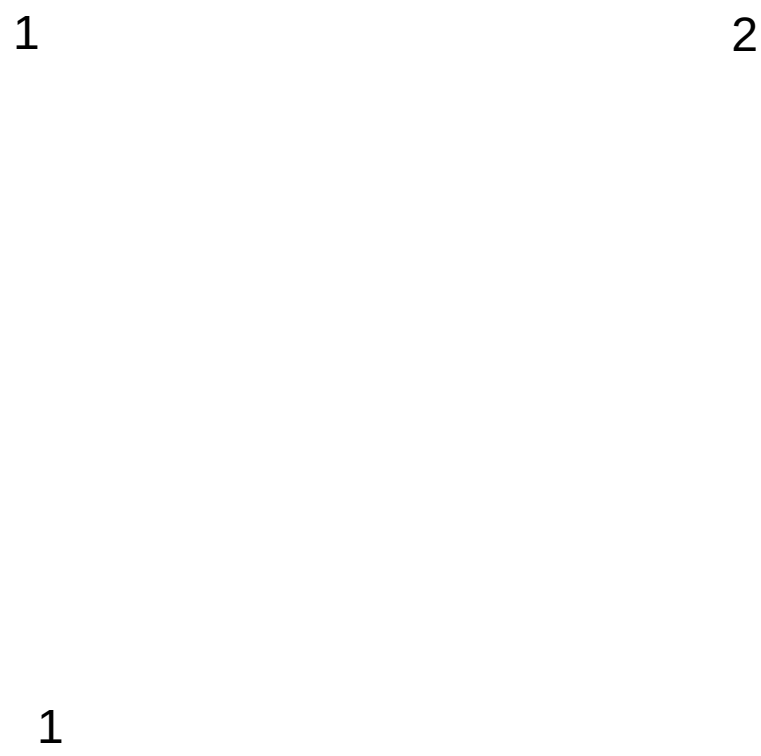
Small caches (32, 64 MiB) quickly run out of free buffers, but are also more easy to synchronize (less stuff to be cached).

Note that when slrun becomes slruN (2/3 of cachesize), the cache is saturated.

For large caches (> 200 MiB), you can increase Minfree and MaxFree to prevent synchronization problems that arise during sudden cached IO.



More free buffers also allow for longer synchronization times.



Cache dynamics: eCS loads a VPC image (1) and saves it again (2).