LBL-15552
c.2

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

Engineering & Technical Services Division

RECEIVED
LAWRENCE
BERKELEY LABORATORY

MAR 21 1983

LIBRARY AND
DOCUMENTS SECTION

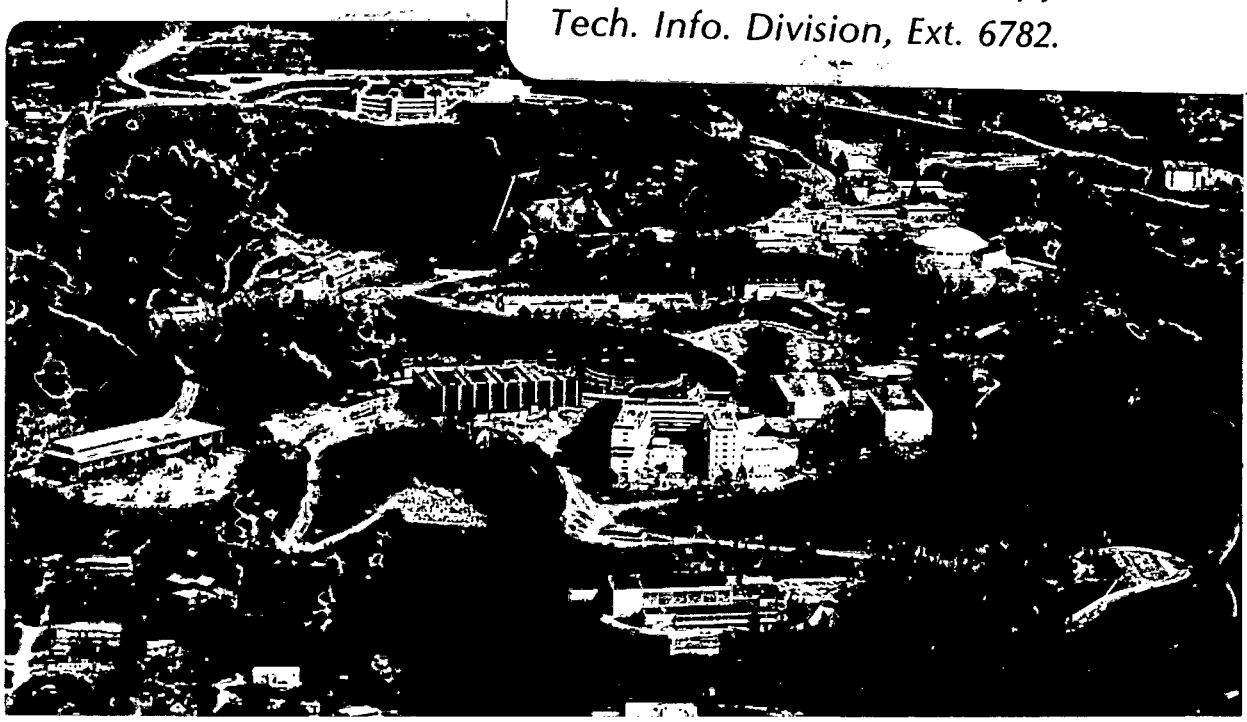
A MAX-IV DISC CACHE

Van Jacobson

October 1982

TWO-WEEK LOAN COPY

This is a Library Circulating Copy which may be borrowed for two weeks. For a personal retention copy, call Tech. Info. Division, Ext. 6782.



LBL-15552 c.2

Prepared for the U.S. Department of Energy under Contract DE-AC03-76SF00098

DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

A Max-IV Disc Cache

Van Jacobson

Real Time Systems Group
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

This note describes some design considerations for routines to "cache" sectors of disc in main memory on a Modcomp-IV or Classic running Max-IV. Anyone who has used the Tools under Max-IV or has a real time system which does a lot of disc I/O (like the NBS system) is aware of the pathetic disc performance. The CPU utilization, even with 5 or 6 users, rarely exceeds 30% (limited by disc saturation). The reasons for this poor performance are probably:

(1) A small disc block size (256 bytes). This generates a lot of I/O requests. (most machines consider 512 to 1K bytes the absolute minimum block size).

(2) The worst possible disc layout. All of the space for file system data structures is allocated when the disc pack is "labelled". This space must be contiguous and must be relatively large (since when you run out, the disc is useless). Thus you negate the "clumping" effect of a reasonable space allocator and guarantee long seeks (from the data space to the file system space & back).

(3) Coarse space allocation granularity. The minimum practical allocation unit is a track. This guarantees that a seek must be done for every context switch (2 users' files can't share the same track). Measurements indicate that heavily used files are small (the "dynamic" average file size is 4-6 sectors) and several could share one track.

(4) Bizarre file system data structures. All the file system internal data structures are built of linked lists of discontiguous sectors. Thus, even on discs with hardware buffering, the file system internals can't take advantage of it. Also, the data structures weren't designed for speedy operation (they probably weren't designed at all): It takes a minimum of 2 disc accesses to look up each simple name of a file in a directory. It typically takes 8-10 disc accesses to open a file.

(5) Few 'in-core' file system data structures. Virtually the only file system info maintained in core is a pointer to the root directory.

The idea of a cache is to use memory & CPU cycles to reduce the number of disc seeks. As the number of seeks goes up, the system performance degrades dramatically. For example, say 2 processes each need to read 4 sectors from different files. If the processes execute sequentially, each requires

```
28ms seek + 8ms rot. = 36ms
```

for the 1st sector & 100μs for each remaining sector. If they exactly interleave their disc requests, a seek has to be done for each read and the total time to execute goes from 72ms to 288ms – a factor of 4 increase to accomplish exactly the same result. A similar non-linear degradation happens when the average seek length is changed – a short seek takes 10-15ms,

This manuscript was printed from originals provided by the author.

a long seek 50-60ms.

1. Considerations

We have 2 types of discs and the cache must work with both. Since they both have the same sector size, this isn't a big deal: there's nothing in a caching algorithm that cares about the disc hardware – it only needs access to an I/O operation at appropriate points. The disc parameters are:

Type	*Size*	*Avg Seek*	*Max Seek*	*Avg Rot.*	*Xfer Rate*
4138	80mb	28ms	56ms	8ms	800kbs
4134	40mb	32ms	58ms	16ms	300kbs

The 4138 controller also includes a buffer capable of holding 1 track's worth of data so successive reads from the same track go at the transfer rate (roughly 100μs / sector). The Modcomp I/O channel is essentially a separate computer with its own port to memory, so disc tranfers cost nothing in terms of CPU cycles.

In general, if one plots system performance vs. cache size, the curve is "S" shaped. For a small cache, sectors are replaced before they have a chance to be re-used and performance is degraded (by the time necessary to copy sectors into the cache). At some point, the time saved from getting high use sectors from the cache exceeds the time spent putting data in the cache and the performance improves. The rate of improvement depends on the relative sector access frequencies (if a few sectors are accessed a lot, the improvement is more rapid) and the cost (in cpu cycles) of maintaining the cache. At some point, all of the frequently used sectors are coming from the cache and there's nothing to be gained by increasing its size. Inspection of the system code & Bob Belshe's disc trace both suggest that the minimum cache size is 100 to 200 sectors. Rule of thumb (from Coffman & Denning's *Operating System Principles)* would say the maximum will be about 400-500 sectors. These numbers obviously have to be determined after the cache is installed but these bounds give guidance in selecting an algorithm (e.g., if the lower limits were 20-30 sectors, one would be thinking of putting the buffers in map 0 or 7 and using simple arrays with linear or binary search for the data structures. For 400 sectors, one thinks of buffers in actual memory, linked lists and hashing).

This brings up an important point: the caching has a definite cost (in terms of system performance). It takes memory (but we have enough so there should be no impact) and it takes CPU cycles. If everything comes from the cache, one simply trades disc saturation for cpu saturation. There's a gain due to the shorter "seek" time of sectors in core but there's a loss of the parallel I/O processor. If the system does enough multi-tasking or is close to CPU saturation, this loss can be significant. I.e., if there was always a task that needed to do 30ms of computing during someone else's seek, there would be no need for a cache).

The two big users of cpu cycles in the caching process are searching the cache for a sector and copying sectors to/from the user from/to the cache. The tightest search loop one can do a Modcomp-IV takes about 10μs / comparison. To search a 400 sector cache linearly would take at least 2ms av. Any algorithm that orders the sectors to get log(n) search time is going to suffer from the frequent re-orderings required as the LRU algorithm adds and deletes sectors (one should expect about a 50-60% hit rate in a well-tuned cache). Since we're throwing memory at this problem anyway, we might as well pick some sort of

hashing algorithm to find the sectors: a uniform hashing function with a 200 word index table should give <100μs search times on a MC-IV.

The other cpu eater, data copying, is particularly nasty. The fastest that one can move 128 words on a MC-IV is 218μs. This isn't bad but, unfortunately, the instructions to do this only work in your virtual address space. The cache is required to move data between 2 different virtual address spaces: the system's (where the cache code & disc handler are executing) and the user's (where the I/O buffer is). This restricts the set of available instructions & requires adding some context switching instructions to the loop. The most straight forward loop (the one in the existing cache) takes about 900μs. Being clever, one can get this down to about 400μs. By doing things which are almost incomprehensible and taking some liberties with a system data structure, one can get this down to about 250μs.

Since the use of the cache has to be indivisible and Max-IV has nothing resembling a semaphore, all of the cache operations either have to be done at interrupt level or with interrupts locked out. Thus 900μs copy + 100μs search means we'll be locked up at interrupt level for a milli-second and may start dropping characters from 9600 baud async lines (which have to interrupt every character). This is probably unacceptable. The 400μs loop might work – it will probably be the first cut.

A last consideration: what units to cache in. Various alternatives are sectors, tracks, cylinders or user defined "chunks" (like the large core image records on a TOC file). The disc trace shows the (dynamic) average file size to be 4 sectors (statically, the average size is more like 12 sectors – apparently a lot more I/O is done to short files than long files. This is reasonable if you consider that :AR and :VA are read by every tool and many of the pipe temp files are short). Since files always start at track boundaries, a track cache will waste 90% (38/42) of its space on the average – we don't have that much memory. In addition, to read those 4 sectors, we had to open the file and that took 8-10 accesses to the management space (250% overhead!). Since the management space is organized as linked sectors strewn randomly across track & cylinder boundaries, a sector cache is obviously optimal for it.

The last possibility above, TOC files, aren't good candidates for the cache: They are read via a single read (like .tsk files in RSX-11M & .exe files on the Vax) and their average size is about 40kbytes. The cache is useful only when the Seek Time to Transfer Time ratio is large. It's at least 10 times *slower* to copy data from the cache than to read it from disc (ignoring seek time). Reading 40kb from the disc will take 25ms (seek) + 50ms (xfer) = 75ms. Reading it from the cache will take 500ms of CPU time.

(Actually, the most compelling reason for using a sector cache is the simplicity of the implementation: the disc handler is, in general, handling user requests to read & write sectors. If the cache can simply steal those sectors from the user's buffer, it will be almost entirely decoupled from the I/O system. If the cache works in units other than sectors, it has to substitute it's own buffers for the user's, then fool the I/O system into putting the appropriate amount of data in the right place. By the time you coordinate this with I/O rundown and swapping, you've either re-written the I/O system or, if you ignore these problems, created a system that maybe works for 10 minutes on alternate Thursdays).

2. The Cache Routines

This section describes the proposed cache routines. Before it can do this, some background on the Modcomp I/O system is needed. The Modcomp has no operating system in the usual sense of an "executive" process which services requests from "user" processes. It is built of 2 classes of things:

- privileged, re-entrant subroutines. The user gets to these routines by executing a trap instruction (a REX) and supplies arguments either by loading the registers or putting them after the REX instruction (the type of system call determines the argument passing mechanism). So far, this looks like a conventional system call. However, on most systems the trap will be serviced by something that has its own identity (e.g., has its own register & memory context) and, by implication, processes requests serially. On the Modcomp, the trap is serviced by a subroutine (e.g., has the same register & memory context as the user). Among other things, this means that anytime that shared data structures (like I/O queues) need to be accessed, the routine has to disable context switching to prevent other routines (or even other incarnations of itself) from accessing the data.

- interrupt service routines. Like the old Sigma-2 system, most of the conventional "system" functions (scheduler, I/O handlers, etc.) are performed in the service routine of a hardware interrupt. To make something happen (like starting an I/O operation), the user (executing one of the subroutines described above) must issue instructions which generate the interrupt. There are 16 interrupt levels (0 = highest priority; 15 = lowest priority) and interrupts are vectored. Two of the levels, 12 & 13, are assigned to I/O devices. The device address determines the interrupt vector address (like on a PDP-11). When an interrupt is active on some level, no other interrupt can occur on that level or lower levels. Level 13 ("Service Interrupt" level) is used for the operating system interface portion of I/O handlers (e.g., I/O request initiation & completion) and level 12 ("Data Interrupt" level) is used for the data transfer portion of handlers that service non-DMA devices (it is unused for DMA devices).

The cache is obviously a shared data structure. The cache code could go 2 places: in the subroutine portion which queues the I/O operation & triggers the interrupt to start the handler or in the Service Interrupt portion of the handler. If it goes in the subroutine portion, it has to execute with context switching disabled and will not be able to put data read in "quick mode" into the cache (you're out of the subroutine by the time the data arrives). Since the main use of the subroutine phase is to gain context switching, the cache code might as well go in the interrupt portion of the hander. This lets it get at all the I/O requests and, since interrupts are serial, removes problems of the cache being 'shared'.

There are several possible states for a handler to go through. The two interesting ones for caching are I/O request initiation and I/O request termination. At initiation, you want to determine if data can be supplied from the cache and, if it can, finish up the request without disturbing the disc. At termination, you want to put the newly transferred data into the cache.

2.1 *The Cache Data Structure*

There are 7 I/O requests to worry about. Of these, two *(read & write)* transfer data.

Four *(advance record, advance file, backspace record & backspace file)* essentially do a 0 word read to find out if there is an EOF mark. The last, *write eof,* puts an EOF mark on a sector. The cache has to worry about 129 words of information for each sector: the 128 words of data and 1 word of 'marks' (EOF, EOI and EOR).

Each I/O request has 2 positions associated with it: the location of the sector on the disc (called the DPI) and the relative position of the sector in the requestor's file (called the FPI). Sectors in the cache are identified by the DPI. To keep big files (which are unlikely to get high use) from evicting useful sectors from the cache, the FPI can be tested to make sure it's below some limit before putting sectors in the cache.

Each sector in the cache has to be accessed 2 ways: via the hash table to see if it exists and to get its data and via the LRU list to evict it to make room for a different sector. In both lists we need to be able to delete nodes from the middle (when we access a sector we want to move it from whereever it is in the LRU list to the front and when we get a new sector (or delete an old one, depending on the algorithm) we need to move it from whatever hash list it's on to the correct one). This suggests that we use doubly-linked lists. The buffers for data should probably sit in actual memory (i.e., not be part of any virtual memory map) since we can map over them whenever we want to move data in and out and, if in a map, the size of the cache is limited by the 64K virtual address space to 500 sectors.

Based on the last paragraph, the cache data structure could look like:

- The DPI of the sector associated with this cache entry. Disc addresses are 24 bits + 4 bits of disc unit number so this entry will be 2 words.
- The hash linkage. A forward pointer to the next entry whose DPI hashed to the same value (0 if none). A back pointer to the last entry whose DPI hashed to the same value.
- The LRU linkage. A forward pointer to an older entry. A backward pointer to a younger entry.
- The buffer address (a 128 word copy of the sector data in actual memory).
- The 'flags' word (EOF, EOI and EOR status for the sector).

All sectors are on both the LRU and hash lists. When things start up (before there have been enough disc operations to fill the cache), all the sectors are put on the 0th hash list with an impossible DPI (-1). The lists are initialized and the memory allocated whenever the system is restarted.

2.2 *The Cache Code*

With the preceding in mind, the cache code looks something like:

On Request Initiation:

```
if sector in cache then

    move sector to front of LRU chain
```

```
if operation is read & wc <= 128  or operation is
   avr or bkr then

    move data & uft status from cache to user
    fake I/O completion
```

On Request Termination:

```
// The criteria for putting the data from the current
// request into the cache are:
//
// All requests: no error
// Reads:        word count = 128
// Writes:       word count <= 128
//
// The ideas behind this are that only full sectors
// are useful (any write is zero padded by the disc hardware to
// the next sector boundary)

if no error & (.( op = read & wc = 128 )
              | (op = write & wc <= 128 ) ) then

    // There's an efficiency possible here since we know
    // that no 'read' type operation is in the cache (if
    // it had been, it would have been picked up at req.
    // initiation & we would never get here).  This means
    // we only have to search the cache for 'DPI' on write
    // and write eof operations.  We assume that the search
    // time is small & don't take advantage of this.

    sector := find sector in cache
    if sector not in cache then
        sector := oldest entry in LRU list
        move sector to appropriate hash list

    move contents of user buffer to cache buffer for sector
    move user's uft status to flags word for sector

else if ( error | wc > 128 ) &
        ( op = write | op = weof ) then

    // since we've just changed some data on disc but the
    // operation wasn't considered ''cache-able'', we want
    // to delete any affected sector(s) in the cache so
    // the contents of the cache will always agree with the
    // contents of the disc.  More than one sector may
    // be affected if the word count was > 128 words.

    for each sector of operation do
        if sector in cache then
            evict sector from cache
```

3. Post-Implementation Notes

The cache was implemented pretty much as described above during the '81-82 Christmas break. The program name, conforming to our 'alphabet soup' naming convention, is FHL. It is about 800 lines of assembly language (this is twice the size I'd hoped it would be) and adds about 1K to Map 0 (800 words of code & 200 of tables). The 2 disc handlers, MH.HAN and LD.HAN, had to be modified to call the cache code at request initiation and termination – about 10 lines of code were added to each handler.

3.1 *Changes to the Design*

The initial intent was to put the cache data structures in Map 7, sharing the map with the File Manager code. Unfortunately, Map 7 on a Classic has only a 16K address space and this isn't large enough (it takes 8 words for a sector descriptor and we want to cache about 500 sectors = 4K. The File Manager code is 13K.) With some trepidation, I decided to give the cache Map 6. This is another useless 16K map on a Classic and, at any rate, Max-IV uses the maps so badly that taking another one has no noticeable impact.

As long as there was a map available, it seemed like a good idea to do the best job possible on the data copy loop. Three pages at the top of the map are used to map over the cache buffer (1 page) and user buffer (2 pages since the 128 word buffer can span 2 different pages). The code to do the map-over and copy is fairly involved but very fast: it measures out at 320μs to copy 128 words on a MC-IV/35 *including* all the setup and mapping time. (This was measured by putting instructions that output 'signature' levels to an IOIS DAC at appropriate places in the code, then measuring the resulting pulse widths on a scope. It wasn't deduced from instruction timings.)

It should have been possible to cache any operation that resulted in an EOF: Program's are *supposed* to ignore the buffer contents on a read if the EOF bit is set. (If you cache on a Write EOF you end up with junk in the cache sector buffer because WEOF doesn't have a write buffer associated with it.) Naturally, at least 2 of Modcomp's programs don't follow the rules – LIB and EDI require that data buffer contain *exactly* the characters "$$" then 127 words of zero in addition to the EOF status in the UFT (This apparently has to do with the Paper Tape Reader (!) putting some extra status bits in the data buffer on an EOF). It took 3 or 4 people and several days of looking at revolting assembly language to figure this out and I was too disgusted to do anything cleverer than simply not cache anything on a WEOF, AVR or BKR.

It turned out that filtering out sectors based on FPI wasn't a good idea. It takes a lot of code to get the FPI (it has to come from the task's TAL which is not something you can get to easily from an I/O node) which slows down the cache. More significantly, at NBS the most useful things to have in the cache were the short "database tables" which, since they were on the big partition maintained by the DBF routines, all failed the FPI test.

3.2 *Cache Performance*

A 256 sector cache gets about a 30% hit rate (if more than one user is on the system). A 512 sector cache gets a 65-75% hit rate. Going to 768 sectors didn't improve the hit rate so I left the cache size at 512 sectors. There was no investigation of sizes between 256 and 512.

Of the operations that miss the cache, I expected the majority to be 'writes' (which can never be hits). This didn't seem to be the case: Of the misses, 70% are 'reads', 25% 'writes' and 5% 'weofs'. This might mean that 20% (70% of 30%) of the sectors read are not reused within more than 700 disc requests – a surprisingly high fraction. However, the statistics don't distinguish between "not in cache" misses and "read > 128 words" misses (e.g., the load of a TOC file) – most of this 20% may be from loading programs.

The 'Writes' to 'Weofs' ratio above gives another measure of 'dynamic' file size: Since we almost always write sequentially then terminate a file with an eof, 'weofs' count the number of files written. Thus, the average number of writes per file is around 6 sectors. This may be an underestimate since a Tools "create" does an extra 'weof' when it creates the file but I doubt if it's off by as much as a factor of 2.

Since the File Manager overhead is so high for short files, I had thought that the majority of sectors in the cache would be from the File Management Space. Actually, only about 30% of the sectors seem to be Management Space (see fig.1). Most are either short temporary files ('pipe' files), peoples' :AR (argument passing) and :VA (working directory, etc.) and TOC file LPRs for the "Root Tasks" and the Shell. (LPRs are the parts of a TOC file that describe the resources, entry point, etc., of the module. They are sector sized and good candidates for caching. MORs are the big core-image chunks of a TOC file that are never cached.) Everybody's shell search path file and all of the directories it mentions end up in the cache.

Using a 251 entry hash table for the sector hash gives average queue lengths of 2 entries (at 10μs search time per entry) under load. (See fig.2). The longest queue is 6 entries but 85% have 3 or fewer entries. The periodicity in the queue lengths (fig.3) shows that the division hash is doing a lousy job. This is almost certainly because the sector number is being used as the least significant part of the disc address. Since the average file size is small, only the least signifcant 3 bits of the 8 bit field are used. Since a division hash works best if the values in the least significant bits (up the the hash modulus) are uniformly distributed, putting the track number in the bottom of the disc address would probably cut the queue lenghts by a factor of 2. Since this would pick up <15μs of a 350μs service time, I didn't think it was worth the trouble to change.

3.3 *Bottom Lines*

For NBS, the cache made the normal shot sequence go from 22sec. down to 18sec. This 20% improvement doesn't seem to jibe well with the 60% hit rate but it's the usual performance improvement problem: as soon as you remove one limit, you bump into another. The NBS Classic is *100% saturated* for the entire 18 seconds after a shot. The next improvement in the NBS system will have to come from some selective re-coding for speed.

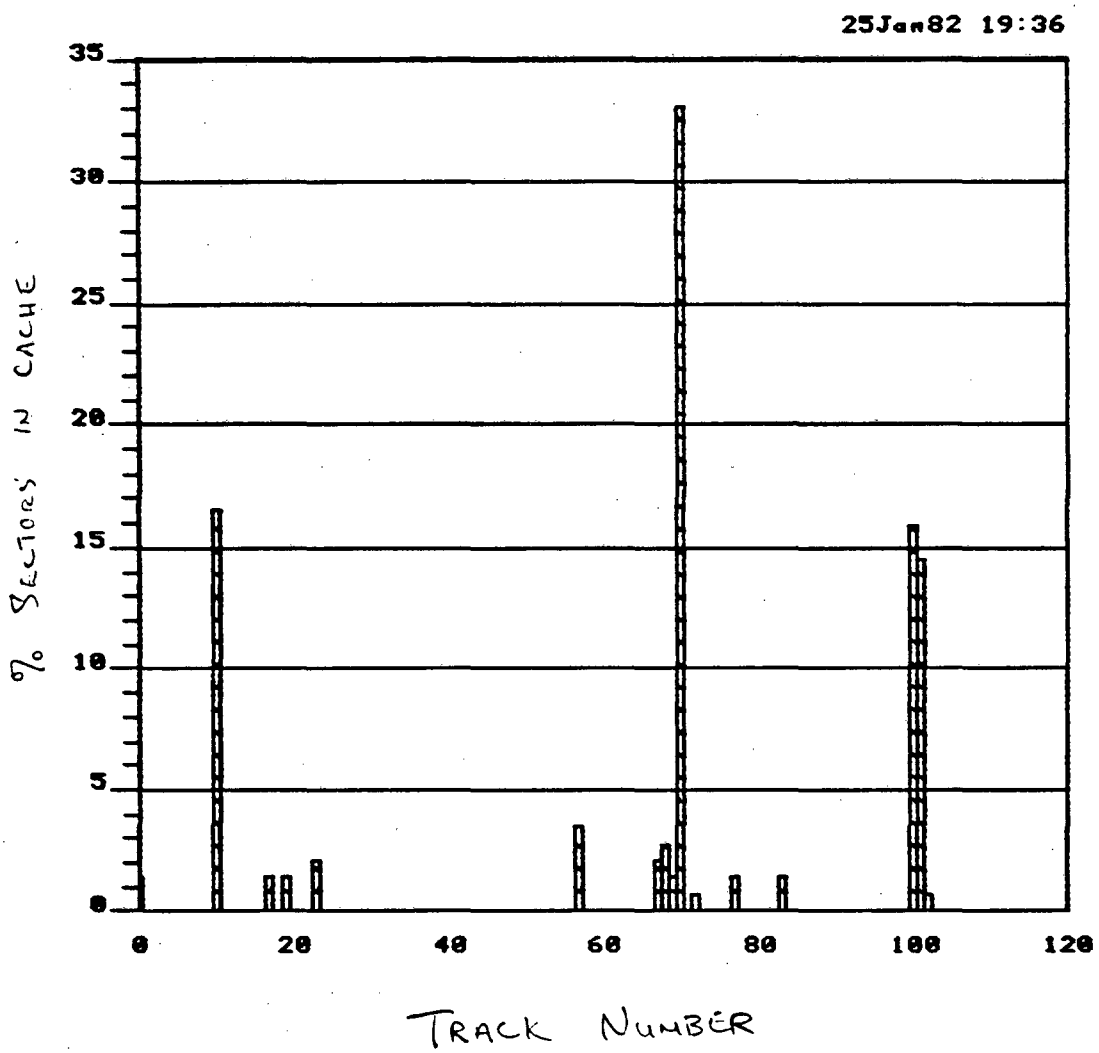
For the Tools' interactive response, there's a similar dismal story. There was a noticeable improvement in several things: "Who" writes out at the terminal speed rather than 1 line per second and the number of directories in your search path no longer seems to have any effect on the response time. However, the bulk of the response is now due the Shell spawns: for every command you execute, the entire 64K core image of the shell has to be written out then, later, read back in. On Dev-II's 2314 discs, this takes *800ms* which, of course, completely overwhelms the cache improvement. The best way to deal with this is probably either to make the Shell smaller or make it sharable. I.e., reduce the amount that must be

read and written on a spawn. (An easier alternative here is probably just to do as much as possible on the Vax).

This work was supported by the U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

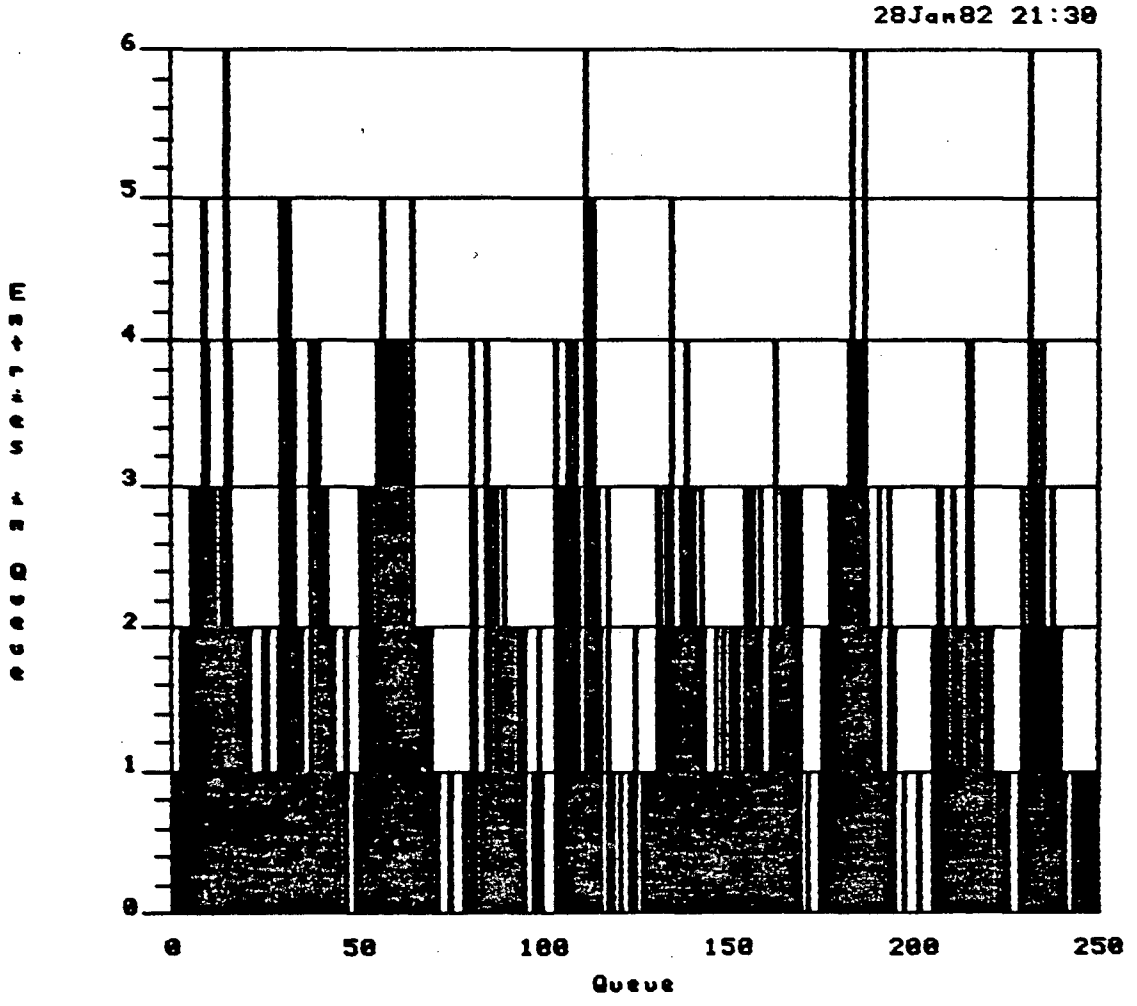
/mnt 145 total

Tracks in Cache for Disc 2



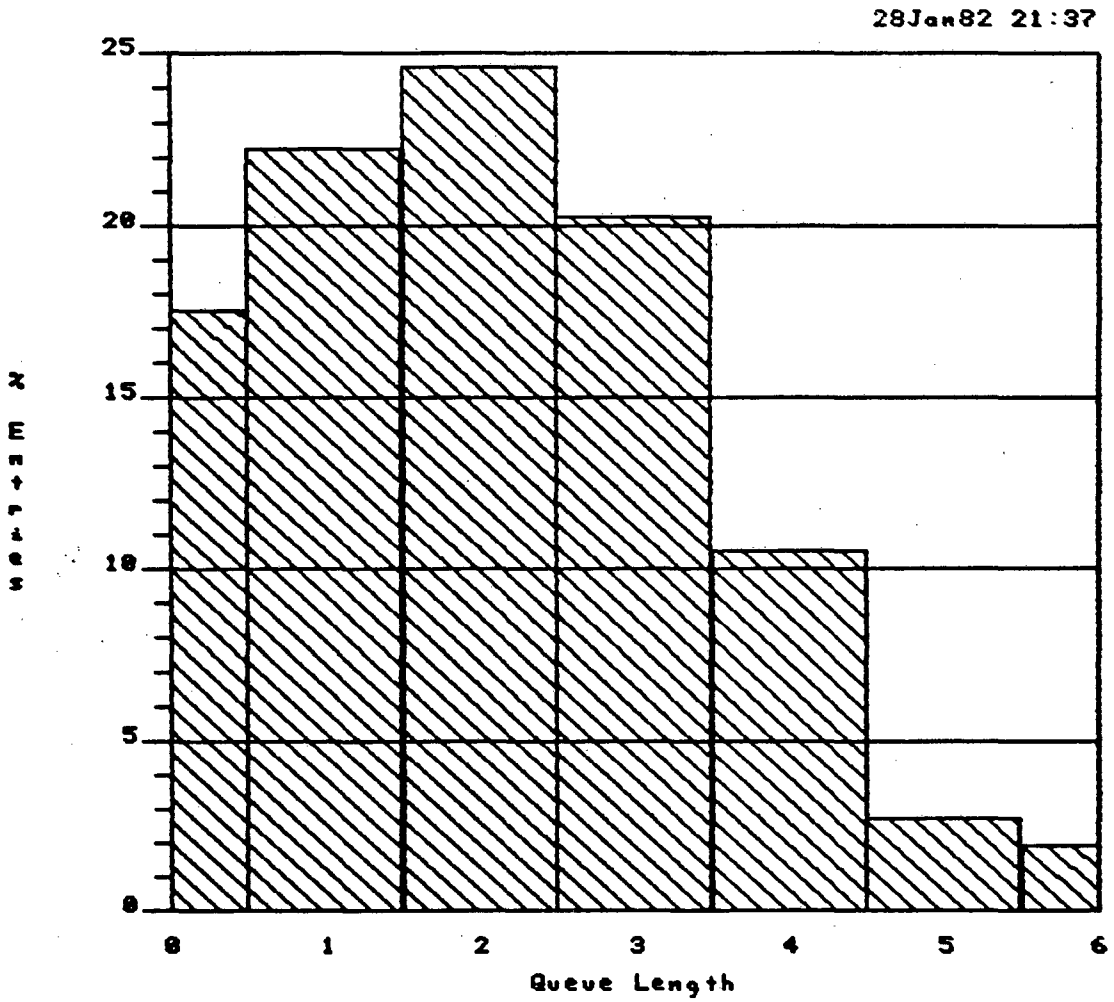
End of day

Cache Hash Queue Lengths

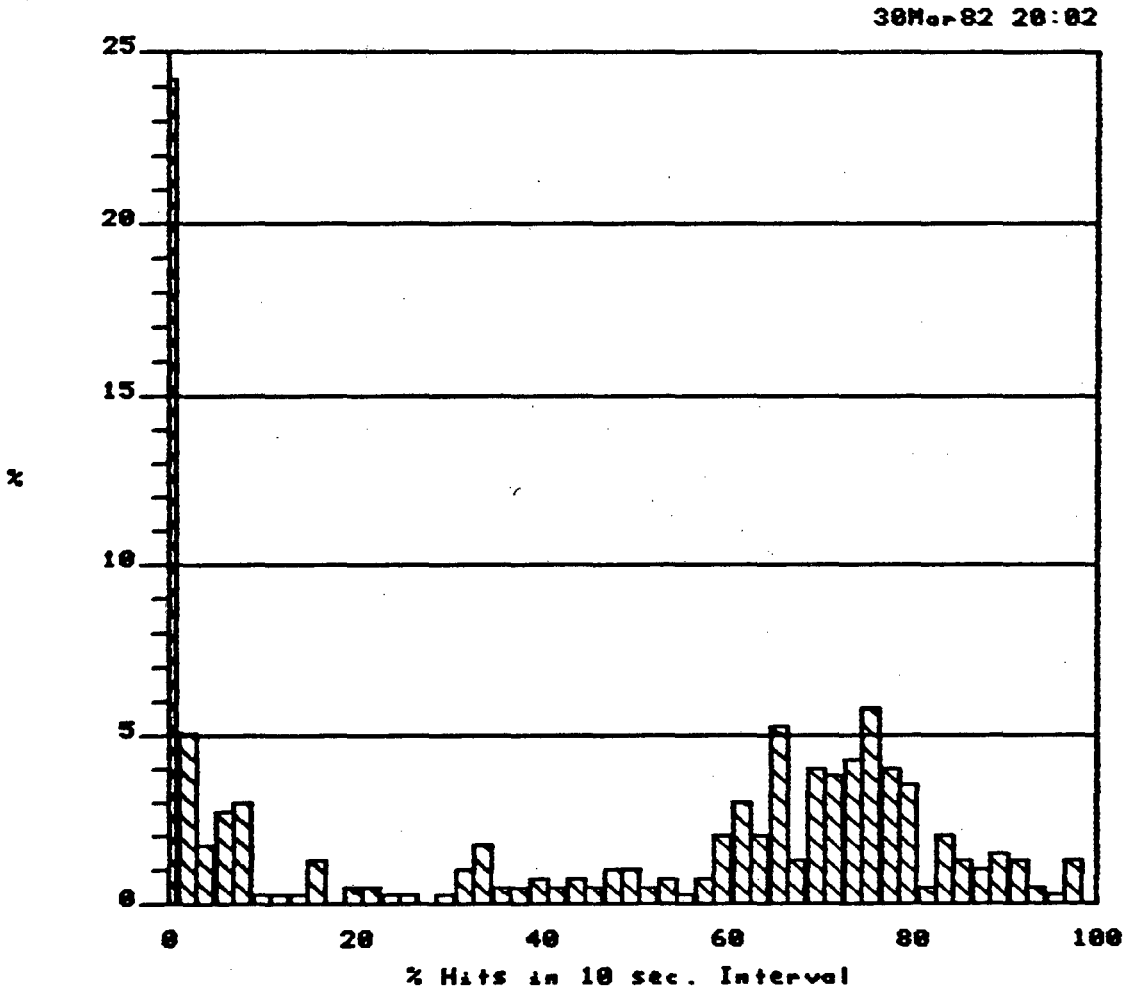


End of day

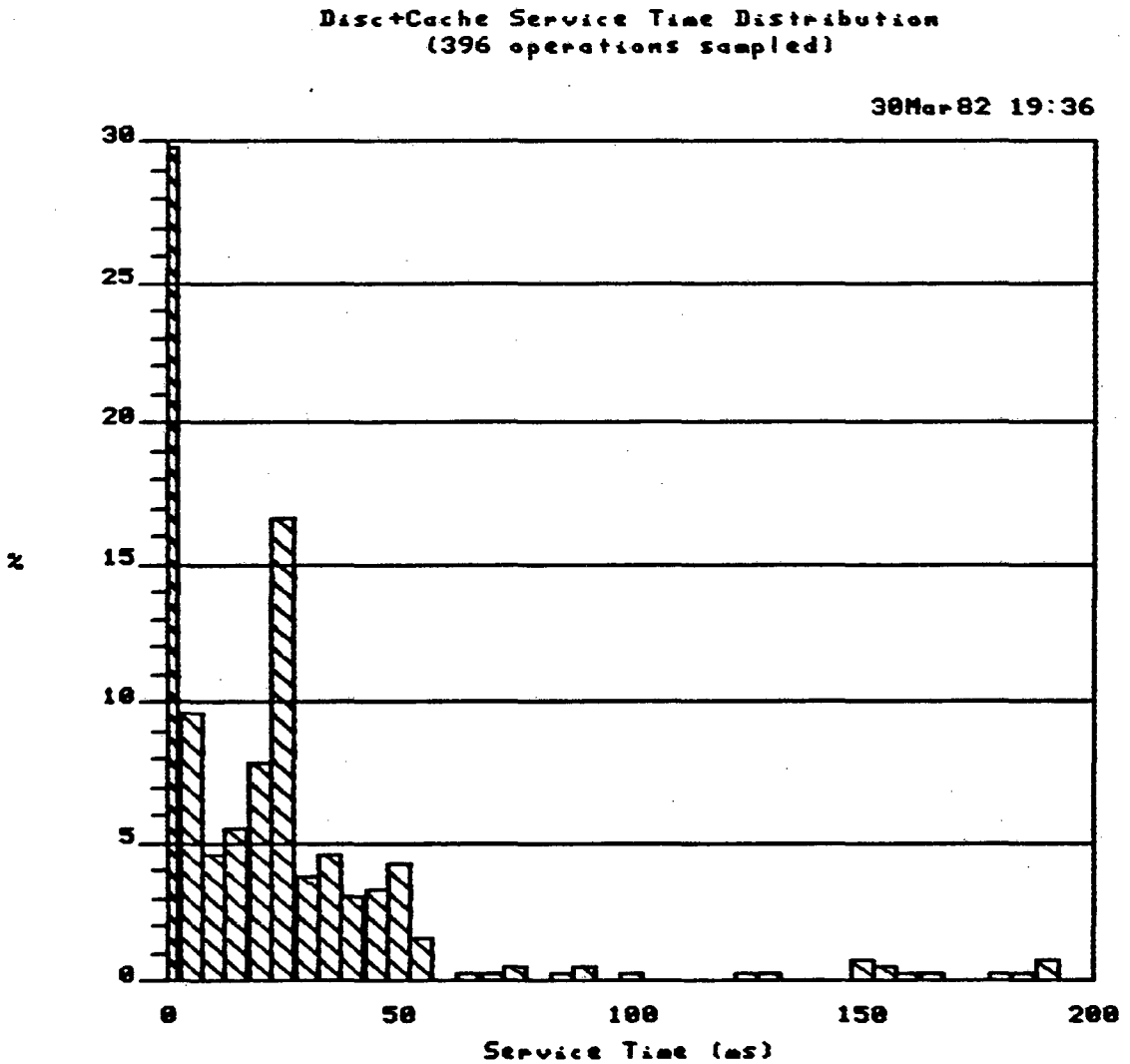
Cache Hash Queue Length Distribution

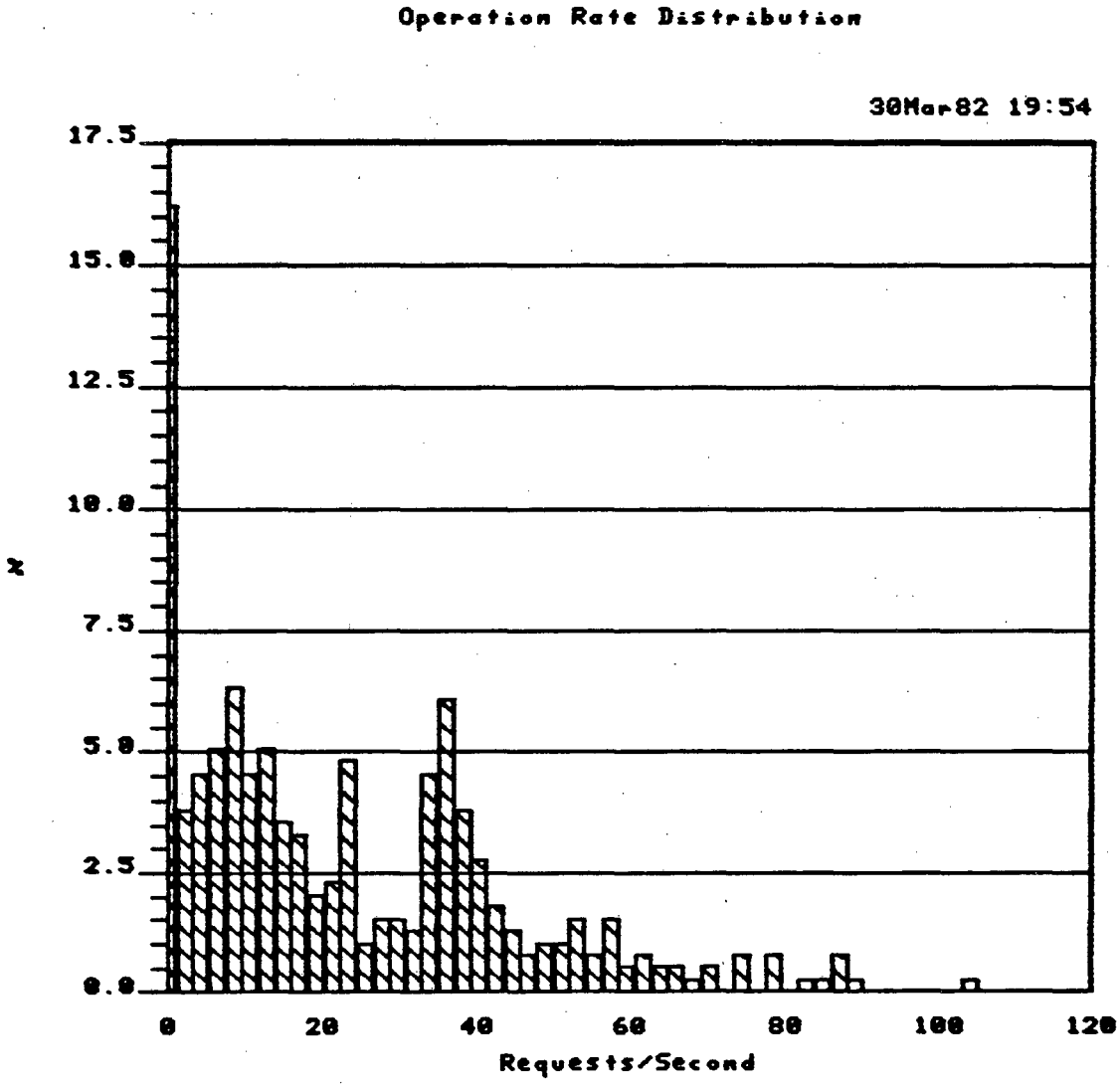


Cache Hit Rate Distribution



2 hour collection
10 sec. sampling
90k Total Disc Ops.
1-3 users




Operation Rate Distribution
30Mar82 19:54
17.5
15.0
12.5
10.0
%
7.5
5.0
2.5
0.0
0
20
40
60
80
100
120
Requests/Second

This report was done with support from the Department of Energy. Any conclusions or opinions expressed in this report represent solely those of the author(s) and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory or the Department of Energy.

Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.

TECHNICAL INFORMATION DEPARTMENT
LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720