ROBELLE CONSULTING LTD.
#130-5421 10th Avenue
Delta, B.C. V4M 3T9
CANADA
(604) 943-8021
Telex 04-352848

HP 3000/OPTIMIZING BATCH JOBS

Technical Report, February 1981, By

ROBERT M. GREEN

Robelle Consulting Ltd.

Summary

The elapsed time of high-volume, stand-alone tasks can sometimes be reduced dramatically. This report is a follow-up to my 1978 paper on optimizing on-line programs. A large number of techniques for batch optimizing are explained, evaluated by empirical testing, and, finally, ranked according to their relative "power". The tests show than some highly-recommended techniques are of little benefit, while other techniques, often overlooked, will make some tasks execute 4 to 12 times faster.

Contents

Section I WHY BOTHER WITH BATCH?

In December of 1978, when I gave my paper in Denver on optimizing on-line programs [15], the techniques that I presented were not common knowledge among users of the HP 3000 (or even among Hewlett-Packard Systems Engineers). The ideas discussed at the meeting ("30 disc I/Os a second" [41]), have spread throughout the user community as a result of numerous papers and articles [39] [48] [09] [19]. With hundreds of intelligent professionals pushing the HP 3000 to support the maximum number of terminals, is it any surprise that many have succeeded?

In less than ten years, the HP 3000 has grown from 128,000 bytes of main memory to four megabytes, and the power of the central processor has increased several times. Software has been rewritten and refined (by Hewlett-Packard and independent vendors) to incorporate the principles for optimizing on-line programs. Today, HP 3000 users develop systems which support 20 to 40 terminals without serious difficulty. In 1973, users had trouble supporting 8 terminals on an HP 3000. The speed of batch jobs, however, is not much better than it was in 1973.

But what about batch processing? One of the most widely recommended techniques for improving on-line response time has been to "dump" big tasks into the batch queue [15] [45]. This is a natural and useful idea; the batch queue is where "batch-type" operations belong. Unfortunately, as applications mature, databases grow in size, batch jobs take longer to complete, and new batch programs are put into production. Eventually, the batch queue is clogged. For many HP 3000 sites, completion of month-end and year-end batch jobs is a major irritant.

Slow batch jobs can be very costly. Operator overtime may be required for a graveyard shift. Extra computers may be needed to handle the batch load. The batch capability of the HP 3000 is not an infinite sink into which "problem" tasks can always be dumped. It is a finite resource that can easily be exhausted.

This report attempts to discover how to complete those high-volume, stand-alone, batch jobs in less time (i.e., 8 hours instead of 16). Although I will note the impact on other users of each optimizing technique, interference with concurrent users will not be a conclusive argument against any proposal. In addition, my focus will be on empirical verification of gains in throughput, with a goal of ranking all optimizing techniques by their relative "power".

This report is very specialized. You will not learn what to do about a flood of small batch jobs that are swamping your HP 3000. (The techniques needed to solve that problem are the same ones that you would apply to on-line sessions: reduce number of logons, eliminate UDCs for batch logons, :ALLOCATE program files, etc.) You will not learn how many batch jobs you should run concurrently. (The answer to that question may change drastically when MPE IV is released.) Nor will you discover how to reduce the impact of batch jobs upon on-line response time. (MPE IV should give you the tools you need to solve that problem, if you are willing to degrade your batch throughput [48].) What you will receive are practical suggestions for improving batch throughput.

This document is the result of an investigation into reducing elapsed time. Many of the conclusions that you will read here were surprises to me, only uncovered when theory was put to the test under controlled conditions. The first three sections of this document are introductory: purpose, theory and method. The fourth section contains the bulk of the investigative results and explanation: thirty proposals for optimizinng. Every technique is given a thorough treatment (although some techniques were found to be much more effective than others). Negative knowledge is as important as positive, if is saves you from wasting time on methods with little chance of success. The last two sections summarize the results: ranking the techniques and showing the implications of the investigation for different readers. Finally, in an appendix, I have listed alphabetically all of the sources that I referenced, plus a suggested reading program for those who would like to become proficient in this subject. The reading list starts with general survey papers and progresses to the most technical reports.

Section II WHAT IS "BATCH" AND HOW FAST SHOULD IT BE?

The distinguishing characteristic of a batch job is that it has an extremely "stupid" controlling terminal. An interactive session is controlled by a terminal with a human operator; the controlling terminal can tell the session what to do if an unusual situation arises. Given the primitive job control language of MPE, a batch job cannot handle many unusual events. Also, the particular batch jobs that this report addresses are the ones that process a high volume of transactions.

BATCH VERSUS ON-LINE

How does high-volume, stand-alone processing differ from on-line, interactive processing? One big difference is that a batch job does not have a user watching its "response time", waiting, hitting the RETURN key, losing patience, and telephone the DP department. If a batch job is slow, no one is likely to complain, and no one is likely to improve it.

We should be able to use the same general strategy to make all programs run faster (batch and on-line), but we may have to vary the specific techniques we use and the areas where we focus our attention. As a start, I would like to review the five "prinicples for optimizing on-line programs" from my previous paper [15]:

"Make each disc access count" is important to both batch and on-line. Efficiency of disc usage may be the determining factor in the speed of batch jobs.

"Maximize the value of each terminal read" is obviously irrelevant; it was designed to "spread out the load" of many independent, unpredictable, terminal users. A batch job is one, continuously heavy demand for resources. All we can do is make certain that we do not run many batch jobs at the same time.

"Minimize the run-time program size" sounds like a good idea; but, if a job is run stand-alone, the program size is not likely to slow it down. In fact, the "tricks" used to reduce the size of a program may actually increase clock time by a small amount (a loss that cannot be measured in an on-line program may be quite noticeable in a batch program).

"Avoid constant demands for execution" is clearly impossible; that is the definition of a batch job. On-line programs, on the other hand, should consist of "short" bursts of activity, divided by long periods of inactivity. If all of the on-line users demanded the resources of the HP 3000 at the same time, and did not let them go, the system would be swamped.

"Optimize for the common events" should apply even more to batch jobs than it does to on-line programs. The events that are repeated thousands or millions of times offer the most potential for reducing overall elapsed time, even if the improvement per "event" is only a small reduction in CPU time.

GENERAL STRATEGY

In theory, there are four strategies that "should" reduce the elapsed time of batch jobs [48] [47]:

1. Eliminate some disc transfers completely.
2. Do the same work with fewer disc transfers.
3. Use less CPU time.
4. Overlap a disc transfer with useful CPU work.

One strategy deliberately missing from this list is: "Minimize the amount of main memory used". Most documents on performance place a high priority on reducing code/data segment size [15] [41] [39] [26] [47] [18]. It is implicitly assumed that you are memory-bound. However, a Series III or Series 44 model of the HP 3000 with full memory is a very large computer. Without using "extra data segments" or other advanced programming techniques, a single program can use only 64,000 bytes of the main memory for data (that may be a small fraction of the main memory that is actually available). Techniques will be examined that trade-off increased code and/or data space for decreased disc transfers or CPU time.

THEORETICAL SPEED LIMIT

The theoretical limits on the speed of a batch job are the physical speed of the disc drives [41] and the power of the CPU. If zero CPU time were needed to process the data (or, if processing could be exactly overlapped with disc transfers), a batch job could progress at pure disc speed.

Hewlett-Packard disc drives have the following speed characteristics [48] [26]:

1. 400 microseconds to transfer a sector of information (256 bytes); 22.2 milliseconds to transfer a full track on the 7925 (16.6 ms. on 7920, 7906, 7905).

2. 11.1 milliseconds average latency on the 7925 (8.3 ms. on the 7920, 7906, 7905); latency is the time it takes for the disc to revolve so that the "head" will be located over the sector where you want to start the next transfer.

3. 5 ms. track-to-track seek time on all drives (25 ms. average seek, 45 ms. maximum seek from edge to edge).

4. 64 sectors per track on 7925 drive (48 sectors on 7920, 7906, 7905).

Since batch processing often involves large sequential scans, we need a target "best" rate for transferring large chunks of contiguous disc data. Assuming a 7925 disc drive and an average latency of 11.1 ms, what is the time needed to transfer 1024 sectors (16 tracks)? The answer depends upon the size of each individual transfer. To understand this point, you need to calculate the effective data rates, using three different transfer sizes: 256 bytes (one sector), 8,192 bytes (1/2 track) or 16,384 bytes (a full track). In all three cases, overall seek time will be about 80 ms., if the data is located in adjacent tracks (5 ms. times 16 tracks).

The time-per-transfer equals the latency time (spin to proper sector), plus the actual transfer time (.4 ms. per sector). The total time to transfer the 1024 sectors of data equals the time-per-transfer multiplied by the number of transfers, plus the seek time (calculated above to be 80 ms.).

Size-of-Transfer	Time-per-Transfer		Time-for-1024-Sectors
256 bytes	11.5 ms.	(x1024+80=)	11.856 seconds
8,192 bytes	22.2 ms.	(x32+80=)	0.790 seconds
16,384 bytes	33.3 ms.	(x16+80=)	0.613 seconds

Why do larger transfers improve the overall data rate by so much (15 to 19 times faster)? Because the disc revolves at a constant rate! If you read one sector at a time, you must still wait for a full revolution of the disc (22.2 ms.) before you can read the next sector.

In practice, full-track transfers are not quite as efficient as they appear above. An application program cannot make use of every disc revolution; it wastes some of them. After reading the first track, your program cannot generate a new read request for the next track before the disc itself has revolved past the starting sector of the next track. Therefore, the best possible data rate is closer to that for half-track transfers than full-track transfers (i.e., .79 seconds per 1024 sectors, not .61 seconds). To this time must be added the unavoidable CPU overhead of MPE (about 8 ms. per request).

CONCLUSION: A rough estimate of the fastest possible speed for batch processing is ONE SECOND PER 1000 SECTORS (see :LISTF,2 for the number of sectors in a file).

Section III HOW TO VERIFY PROPOSALS FOR IMPROVING BATCH

MPE and the HP 3000 constitute a very complicated mechanism. Sometimes, so many factors are at work that an "obvious" truth turns out to be totally false (or so hedged with qualifications as to be useless). Wherever possible, I have tested each optimizing proposal by performing a reproducible experiment, rather than depending upon intuition, common sense or theoretical "proofs". Fortunately, stand-alone batch jobs are the easiest computer tasks to measure.

The key to a good experiment in optimizing is to vary only one factor each time a test run is made. For example, to test the effect of different blocking factors, I have attempted to run exactly the same program on exactly the same data, changing only the blocking factor. This sometimes conflicts with another goal, that of using "real" programs in tests. Real programs may perform other operations that vary from run to run and are not exactly reproducible (i.e., spooled output may go to different positions on different discs).

I have focused on reducing the elapsed time of fixed batch jobs. Normally, the CPU time increases or decreases in tandem with the elapsed time; so I have only shown the CPU time when it is exceptional. Unless otherwise noted, all tests were run on a Series III, under MPE release 2011 ("Athena"), using 7925 discs.

Section IV — POSSIBLE TECHNIQUES FOR BATCH OPTIMIZING

In order to compile a list of suggestions for optimizing batch jobs, I searched through past literature on the HP 3000. Although there are no papers on this exact topic, many papers described specific techniques that can be applied to batch jobs. I reviewed all of the HPGSUG publications (journals, newsletters, conference proceeedings), a large body of Hewlett-Packard documentation (S.E. notes, support newsletters, manuals, etc.), and many private reports that I have accumulated in ten years of working on the HP 3000. I reduced the results to a list of about 30 different ideas.

The optimizing techniques were then grouped into three classes, based upon the degree to which existing applications must be modified:

A) CHANGES TO DATA STORAGE (easiest),
B) SIMPLE CODING CHANGES, and
C) CHANGES TO APPLICATION DESIGN (hardest).

For each proposed optimizing technique, I attempted to perform a verifying experiment. When that was not possible, I reported experiments done by other users. In a few cases, I mention ideas that have not been verified to my satisfaction. These ideas are carefully noted; you, the reader, are invited to send me the results of any tests you may do to prove (or disprove) these suggestions.

A. CHANGES TO DATA STORAGE

If throughput can be increased by changes in the way the data is stored, this will often be the most economical optimizing alternative. Since a data storage change does not usually require any program changes, expensive programmer time is not needed.

A1. INCREASE BUFFERS FOR MPE FILES

MPE disc and tape files are "buffered"; several logical records are packed into each physical record ("block"). The file system reads these blocks into buffers that are kept in extra data segments, then passes individual logical records to/from the program as requested. When you access a file sequentially, the file system "pre-reads" the next physical block. Theoretically, buffering should increase batch throughput, allowing overlap of useful CPU work with disc transfers. However, tests have shown that this is not the case.

In a test of file-copies with FORTRAN [09], two buffers (the default) was 1.08 to 1.16 times faster than one buffer, but only if you called FREAD/FWRITE directly (instead of using the FORTRAN I/O statements). Increasing the buffers beyond two did not improve the speed at all (:FILE XXX;BUF=4).

The time to copy 1000 records (333 sectors) varied from 10 to 28 seconds. This works out to an effective "data rate" of 15 to 42 seconds per 1000 sectors (666 sectors must be processed in total, since each sector must be read in and written out). The "ideal" speed for file copies, as deduced in Section II from the speed of of the disc hardware, is less than 2 seconds per 1000 sectors. (Many times faster; certainly room for improvement!)

In another test [38], sort times were 1.04 to 1.07 times faster with two buffers, instead of one, but only if the blocking factor was small (the test was done using the pre-1918 sort, without the benefit of NOBUF). More than two buffers did not improve sort times. In fact, it sometimes caused a slight increase in times. I ran my own tests with FCOPY, and verified these results (BUF=2 is 1.085 times faster than BUF=1, but BUF=4 is not faster than BUF=2).

Why doesn't buffering work? Another paper [48] explains it very well. MPE III uses 2.9 milliseconds of CPU time for each logical transfer (buffer to stack), plus another 8 ms. for each physical transfer (disc to buffer). For an 80-byte record, blocked 16, the CPU time to read a block is 52.5 ms., but a disc access only takes 33 ms. MPE III IS CPU-BOUND! The author of [48] suggests that buffering can help random access files by creating a "cache" of active logical records! But, the only time I tried this technique, the program ran slower. These conclusions may change on the Series 44 (MPE IV), as tests indicate that FCOPY is 2.25 times faster than on a Series III with MPE III [36]. MPE IV may not be CPU-bound.

A2. INCREASE KSAM KEY BLOCK BUFFERS

KSAM, an acronym for Keyed Sequential Access Method, is the Hewlett-Packard equivalent of ISAM. KSAM uses buffers located in an extra data segment, but differently from the file system. KSAM always allocates one buffer for the data block and a number of buffers for the key blocks. It appears that KSAM now allocates enough key block buffers (in most cases) by taking into account the number of levels in the B-Trees. However, if the KSAM file is empty, KSAM may not allocate enough buffers. In one experiment [10], the user improved the time to load an empty KSAM file by 2 to 10 times through an increase in buffers. (The key block buffers are controlled by the 'numcopies' field; :FILE XXX; DEV=,,13 for 13 buffers.)

A3. INCREASE IMAGE BUFFERS IN DBCB

The IMAGE/3000 database system also uses buffers external to the user data stack to provide blocking of logical entries into physical disc blocks. There is a major complication with IMAGE. Unlike KSAM and the file system, IMAGE uses a shared, "global" buffer pool for each database, with the users of the database competing for the available buffers. IMAGE allocates an extra data segment called a DBCB, large enough to hold the number of buffers that IMAGE thinks will be needed. This number is based upon the highest number of paths into a detail dataset, and the number of users who have the base open (i.e., the number of buffers may vary dynamically during the day). This default number of buffers can be overridden with the BUFFSPECS command of DBUTIL. What a tantalizing prospect!

One published test varied the number of buffers while doing a DBLOAD [49]. When DBLOAD reloads a database from tape, it is acting like a long batch program that does many DBPUT operations. The reload time was reduced 1.26 to 2.53 times by progressively increasing the buffers for one accessor from the default (9) to the maximum (255). The fastest time occurred between 30 and 100 buffers. That is not surprising, since the actual number of buffers that can be allocated is limited by the size of the largest extra data segment (32,000 words). With a blocksize of 512 words, the maximum is about 50 buffers (100 buffers for 256 words, 25 for 1024 and only 12 for 2048 words). DBUTIL does NOT give you a warning if you request more buffers than is possible [07]. I have contributed a program called LISTDBCB that shows the current size of the DBCB of any database (some versions of SOO show the DBCB also).

The one experiment that I did to verify this technique gave disappointing results. By doubling the number of buffers, I only improved the elapsed time (to do 2551 DBPUTs) by 1.03 times. Perhaps my DBPUT operation was not complex enough to require the extra buffers that were available; or, doubling the buffers may not have been enough.

During on-line access to the database, I recommend the default BUFFSPECS. Due to the algorithm that IMAGE uses to allocate buffers to users, it is very easy for the entire buffer pool to be "flushed" [07]. Thus, an increase in the total number of buffers can actually make your on-line response time worse by increasing the size of the DBCB that must be swapped. For stand-alone batch access, it is worth experimenting with increased buffers, especially if you are doing complex transactions involving puts, deletes and updates to several datasets, or to datasets with many paths. I suggest that, for one database accessor, you ask for the maximum buffers (i.e., set BUFFSPECS to 255 and let IMAGE allocate as many buffers as it can); there is no point in choosing a compromise number. Also, both of the tests reported above were run in "output-deferred" mode (see technique B4 below); the results may not be as good with the default "output-complete" mode.

A4. INCREASE BLOCK SIZE OF MPE FILES

The most well-known maxim of optimizing is: "increase the block size for batch (serial) access and decrease the block size for on-line (random) access" [33]. How well does this maxim actually work on the HP 3000? The block size is determined by the blocking factor, which is the number of logical records stored in a single physical block of disc space. A block always starts on a sector boundary and is the smallest unit that can be transferred between disc and memory. Since lack of disc transfers is a primary limitation on throughput, increasing the number of records retrieved in each physical transfer should increase the speed of batch programs.

For MPE files, the blocking factor is determined when the file is created (:BUILD XXX;REC=-80,16,F,ASCII specifies a blocking factor of 16 and a block size of 1280 bytes). If you do not specify the blocking factor explicitly, MPE selects a default value by dividing the record size into 128 words (a sector). If the record size is over 128 words, MPE uses a value of one. Thus, MPE chooses the smallest possible block size. Several programs are available in the contributed library that select an alternate blocking factor to minimize the disc space allocated. Another way to choose the blocking factor is to minimize processing time (but, what block size does optimize for speed?).

A number of published tests have measured the effect of varying the blocking factor. A test of file copies with FORTRAN [09] found that increasing the block size above the default improved speed by 1.25 to 1.53 times, with benefits diminishing when the block size exceeded 512 words. The improvement was more marked when I/O was done using the FORTRAN read/write statements than when FREAD/FWRITE were called directly (perhaps because the base time was much slower). The test was for 80 byte records, blocked from 3 (default) to 32. The test also showed that the default blocking (3 r/b) was 1.8 to 2.22 times faster than one record per block. I reproduced this test using FCOPY instead of FORTRAN and obtained similar results, except that the improvement was not as great.

A test of sorts on disc files (before the 1918 release of NOBUF sorts) showed speed increases of 1.26 to 1.42 times, with no improvement above a block size of 1000 words [38]. In order to investigate the impact of actual block size (as opposed to blocking factor) on performance, I ran some FCOPY comparisons with 150-word records. I varied the blocking factor from 1 (default) to 10 and recorded speed increases of 1.17 times (blocked 2) to 1.28 times (blocked 10), with only a small improvement above 600 words. In no case did I find a significant increase in buffered access speeds by increasing the block size from 512 to 1024 words.

I conclude that there is so much CPU overhead in the file system (and the other general interfaces that sit on top of the file system), that the best you can expect is an improvement of

1.25 times (if your blocks are small currently). Since very large blocks can have a detrimental impact on terminal users and show no performance benefit, you should pick the block size between 512 and 1024 words that minimizes disc space. However, it is also possible to access files in a non-buffered mode (see ideas B2 and B3) for better performance. I will show that it is possible, with NOBUF access, to have "the best of both worlds" (big block for batch and small block for on-line), if you choose your block size correctly.

A5. INCREASE BLOCK SIZES FOR KSAM

In KSAM, there is both a data file blocking factor and a key file blocking factor [10]. The blocking factor of the data file should be chosen using the same guidelines as for an MPE file. The blocking factor for the key file "should" impact performance (according to the KSAM manual). I haven't used KSAM enough to deduce a better method for selecting this number than that used by KSAM itself.

Each KSAM user has a separate, extra data segment for each KSAM file opened. Increasing the block size (or the number of key block buffers) will also increase the size of these extra data segments, and may have a dramatically bad effect on terminal response time. (The same thing is true of regular MPE files as well, but not of IMAGE.)

A6. INCREASE BLOCK SIZE OF IMAGE DATABASE

The dataset blocking factor of an IMAGE dataset is comparable to the blocking factor for MPE files and KSAM data files [48] [33] [07]. The largest block size allowed in a database is determined by the $CONTROL BLOCKMAX command in the schema file, with the default value being 512 words. Given a "target", IMAGE chooses the smallest block size within the target that mimimizes the amount of disc space for the dataset. Some datasets may be assigned a block size just below 512 words and others a block size just below 384 or 256 words. The buffers that are allocated in the DBCB, however, are all the size of the largest block in the database. Therefore, if you have a single dataset with a block size of 2048 words, you are limited to 12 or 13 DBCB buffers, even if your other blocks are only 1024 words long. This will not only decimate your terminal users, but may even degrade batch jobs which perform complex database transactions.

I did a large number of sequential extract tests on a dataset with 105,000 records, using a block size of 256 words, 512 words and 640 words (I wanted 1024, but IMAGE chose 640 instead). The 512-word blocks could be scanned 1.05 to 1.24 times faster than the 256-word blocks. Blocks of 640 words were slightly faster than 512. Given the problems for on-line users, it appears that the IMAGE default BLOCKMAX is optimum.

In these database extract tests, the "data rate" varied from a low of 56 seconds per 1000 sectors (using QUERY on 256-word blocks) to a high of 29 seconds per 1000 sectors (using * field list, dataset number and 512 word blocks). For a database extract task, the ideal data rate is 1 second per 1000 sectors. Why was our rate at least 29 times slower? There are two reasons: IMAGE needs CPU time to check and execute each intrinsic call, and IMAGE reads only one data block per revolution of the disc. See topic B2 below for a method of bypassing the CPU time of IMAGE, and topic B3 for a method of retrieving several IMAGE disc blocks per disc read (up to a full track of information).

A7. IMPROVE HASHING OF IMAGE MASTER DATASETS

The topic of "hashing" in IMAGE master datasets is well covered in other documents [15] [45] [07] [30] [b10] [34] [33]. I suggest that you run the contributed program DBLOADNG [07] at least once a month. Look for a high percentage of "inefficient pointers" in your master datasets (over 20% means you are often using more than one disc read to locate a record), and a high "elongation" (over 1.5 indicates a problem). Any improvement in the hashing of master datasets will benefit both batch jobs and on-line programs.

What can you do to improve hashing? Either increase the block size, reduce the record size, or expand the capacity to a higher prime number. How much will this improve the speed of your batch jobs? I have not yet run a controlled experiment. However, if you have a "regular" problem (dataset too full or block size too small), elimination of this problem "should" improve batch programs (that do DBFINDs or Mode-7 DBGETs) by about 1.25 times. If you have a "serious" problem (clustering of records) caused by a bad choice of key field type (and/or values), solving the problem (by converting to a different data type and/or restructuring the key values), will bring a dramatic improvement. I have seen cases where the elapsed time to do a single DBPUT was between 30 and 60 seconds (due to extensive clustering), when it should have been less than a second.

A8. DBLOAD DATABASE TO OPTIMIZE PRIMARY PATH

If you have an IMAGE detail dataset with 11 records per block, and one of the paths has an average of 11 records per chain, how many disc reads will you need to retrieve the entries on a given chain (not counting the hash to the master dataset)? Did you say "one read"? That would only be true if the dataset were perfectly "organized". In the real world, entries are added to and deleted from chains in a "random" fashion, creating holes that are reused for other chains. Over a period of time, the chains in a dataset tend to become disorganized (assuming puts/deletes). Therefore, it may take as many as 11 disc reads to retrieve all of the records on your "average" chain, or as few as one.

The contributed program DBLOADNG reports on the randomness in the chains of each master-detail path [07]. The last column in the DBLOADNG report is called "elongation" and is the critical value. If elongation equals 1.0, the chains for that path are as well-organized as is possible (given the blocking factor). If elongation equals 5.0, then those chains are unorganized; they are spread among five times more disc blocks than is theoretically necessary.

What can you do to reduce "elongation"? Very little, unless the disorganized path happens to be the PRIMARY PATH for the dataset. In that case, reloading the database will repack the entries so that elongation for that path is close to 1.0 [45] [34]. Of course, this only fixes the primary path, not the other paths into the dataset. Therefore, the primary path for a dataset should be the actively-used path with the longest average chain length (you cannot optimize a chain with only one entry!).

In order to "reload" a database, you must DBUNLOAD it to magnetic tape, erase the database with DBUTIL, and DBLOAD the data from the tape. I reloaded a database of 75,000 sectors (19.2 megabytes) which had undergone many DBPUTs and DBDELETEs without a reorganization. DBLOADNG showed that elongation for the largest dataset (capacity 200,000 entries) was 5.85 (very bad)! The DBUNLOAD/DBLOAD took two hours and, as predicted, elongation dropped to 1.17. The average blocks per chain was reduced from 8.3 to 2.3, over 5 times better.

I reloaded the database again to double the block size; this further reduced the blocks per chain to 1.6. If chained access is very frequent, the larger block size, combined with regular reloads, should improve batch speeds (as well as on-line response). I have not had time, however, to verify this hypothesis on an actual application.

A9. RELOAD SYSTEM TO REDUCE DISC FRAGMENTATION

Many sources suggest that you keep at least 20% free disc space (15,000 sectors minimum), and that you not let the number of entries in the free space tables get too large [23] [39] [41]. This is accomplished through a RELOAD from magnetic tape (a full backup). I am not convinced that this will make any difference in the speed of batch tasks. However, my personal experience does verify that disc fragmentation and lack of free space lead directly to increased System Failures. Since System Failures obviously reduce system throughput (on-line and batch) [06], I have no hesitation in suggesting that you do frequent RELOADs.

A10. CONTROL FILE PLACEMENT FOR "HEAD LOCALITY"

The favorite strategy in the optimizing literature is called "head locality" [48] [23] [15] [47] [41] [42] [07]. By placing files carefully on selected spindles, we ought to be able to keep the arm from moving back and forth so much. According to this theory, "head locality" plays a big factor in performance by reducing the average disc access time. It is frequently suggested, for example, that master datasets should be placed on different drives from their detail datasets, because the two are often accessed at the "same time".

The only tests of this technique that I have uncovered [38] [04] do not demonstrate a universal benefit from head locality. The first test attempted to improve sort performance by placing SORTSCR on specific drives. Unfortunately, sort times were actually slightly faster when all three files (input, output, scratch) were ON THE SAME DRIVE. In the second test, the user achieved a 6% improvement in his batch jobs after moving datasets to separate drives. This is not much improvement, considering the inconvenience and time required to obtain "head locality" (:STORE, :RESTORE, or copy from disc to disc). In the final test, the user improved file copies by 1.14 to 2.3 times when he copied from one drive to another [13]. However, a straight file copy is much simpler than most actual application programs.

I was concerned that these inconclusive findings might be "flukes", caused by faults in the experiments. Therefore, I devised several experiments to prove that "head locality" would pay big dividends. First, I reproduced the sort experiments referred to above; the sort times were essentially EQUIVALENT, regardless of where the input, output and scratch files were located. I tested file copies, extracting 10,000 records from a file with 100,000 records. When the files were on separate drives, the copy was 1.08 times faster than when they were on the same drive. This was not very encouraging. Finally, I tested test the effect of separating masters and details. I read 2000 records from a disc file and DBPUT them to a detail dataset that was indexed by one master dataset. When all three files involved were carefully isolated on three separate drives, the extract task ran slightly SLOWER than when all three files were moved to the same drive. This was a most suprising result.

The situation with masters and details may be complicated by the fact that DBPUT and DBDELETE must update the "userlabel" on each dataset (to record the number of available entries), and this label is always located at the beginning of the dataset. Perhaps the steps involved in achieving "head locality" (:STORE, :RESTORE) reduce the overall efficiency of the system by fragmenting the free space. These results show that MPE is seldom as simple as it seems. "Head locality" deserves more research, especially on MPE IV. Until that research is completed, I suggest that users discontinue efforts to obtain head locality.

B. SIMPLE CODING CHANGES

A "simple coding change" is one that changes the method of a program (the "how") without changing the purpose of the program (the "what"). The techniques proposed here can be implemented by mechanical changes to programs (i.e., code substitution). In fact, most of these ideas could easily be provided by the language subsystems automatically (e.g., by the COBOL compiler). Since the objectives of the application program are not modified, only programmer time, not system analyst time, is needed.

B1. USE MORE EFFICIENT PARAMETERS ON IMAGE CALLS

Another technique that is universally recommended is to code your IMAGE database calls with the "best" parameters [01] [45] [07] [30] [34] [etc.]. For the field list parameter (which fields within the dataset to process), the source with the most detailed testing [01] concluded that write access, plus the "@" list (all fields) is always the fastest. However, write access, plus a field list that is subsequently replaced by an "*", is almost as fast. Tests that I performed (doing DBGET extracts serially from a large dataset, varying the field list parameter) verified these results. When there were only four fields in the dataset, @ improved DBGET speed by 1.34 times, and "field-names-plus-*" improved DBGET by 1.33 times. Since the @ option can lead to program maintenance problems, I suggest the use of field names (only the fields you need) followed by an *. I also checked use of the dataset number in place of the dataset name, and found the improvement to be too small to be worth the trouble (861 seconds instead of 869).

Using the optimum parameters in IMAGE intrinsic calls improved the "data rate" for database extracts from 45 seconds per 1000 sectors of data (using field names) to 29 seconds per 1000 sectors (using *). In order to get closer to the "ideal" data rate of 1 second per 1000 sectors, we must use another approach which is far faster than any DBGET call: bypass the normal database overhead completely for high-volume serial access by reading the data directly using NOBUF. This technique will be explored next.

B2. USE NOBUF ACCESS TO REDUCE CPU TIME

The most powerful technique for making batch jobs faster is NOBUF access. Normally, files are accessed through intermediate buffers provided by MPE, KSAM or IMAGE (as described above under A1-A6). However, in my 1978 paper on optimizing [15], I described how you can disable buffering using the NOBUF bit (in FOPEN) and read physical blocks directly into your data stack. When you use NOBUF, it is your responsibility to "deblock" the logical records from the physical blocks. At the time, I was discussing the application of NOBUF to on-line problems; specifically, how to write a program development editor that would not be a drain on the system. The resulting program, called QEDIT, used NOBUF (and other techniques) to cut the load of editing at least in half, while still providing editing speeds that were 2 to 12 times faster than EDIT/3000.

Encouraged by the success of NOBUF in QEDIT, I wrote another program in 1978, called SUPRSORT, that used NOBUF access for file copies and file sorts. This program proved to be 2 to 10 times faster than FCOPY and 2 to 5 times faster than SORT/3000, primarily due to use of NOBUF (see topic B3 below for more information). NOBUF works so well because it REDUCES CPU TIME DRAMATICALLY, by eliminating calls to the file system (or KSAM or IMAGE) and by replacing the general-purpose "deblocking" code of those systems with specialized deblocking code written by the user.

Since 1978, the "secret" of NOBUF has spread from user to user and has been explained in many papers [09] [08] [29] [11]. One of Hewlett-Packard's System Engineers wrote (and contributed) FASTIO, a set of general-purpose deblocking routines that allow the COBOL programmer to benefit from NOBUF easily [44]. Users then reported results such as a reduction in CPU time from 76 seconds to 10 seconds, a reduction in elapsed time from 7.5 minutes to 1.3 minutes, and scan rates of 60,000 records per second instead of 8500 [03] [02].

FASTIO was designed to accelerate sequential access to files; but, an enterprising vendor wrote another set of deblocking routines (called BREAD [26]), that provides both random and sequential NOBUF access. BREAD is callable from BASIC, as well as COBOL, SPL, and FORTRAN and it has better documentation than you can expect from a contributed routine like FASTIO. Using either FASTIO or BREAD, a batch application file can access disc files 3 to 20 times faster than by using the standard READ/WRITE statements provided by the programming language.

With the 1918 release of MPE in 1980, Hewlett-Packard upgraded SORT/3000 to use NOBUF. (There can be problems with NOBUF if you are not careful. Witness the failure of the 1918 sort release to work with card input files or line printer output files.) With these enhancements, SORT/3000 is now slightly faster than SUPRSORT at sorting MPE disc files (up to 10%).

A user who ran timing tests on a file of 50,000 records (64 words long, 20 records per block), found that a NOBUF sort (using either SUPRSORT or the 1918 version of SORT/3000) ran slightly faster than a simple FCOPY of the same file (he also found that SUPRSORT copied this file four times faster than FCOPY) [36]. In other words, the MPE deblocking of 50,000 records took the same amount of time as a sort of the same records. At another site, a junior programmer wrote a set of deblocking routines for magnetic tape blocks (slightly more complicated than disc blocks). After modifying several report programs that scanned tapes, he was able to achieve a ten-fold reduction in CPU time and two-times-faster elapsed times.

Since NOBUF works by reducing CPU time, any changes in MPE to make the file system faster will reduce the "perceived" benefit of NOBUF. This has happened with MPE IV and the Series 44. One of the test sites for this new Hewlett-Packard product found that FCOPY runs twice as fast as it did on the Series III with MPE III [36]. The NOBUF copy was now only 2 times faster than FCOPY, not 4 times.

NOBUF can also be applied to KSAM files, but with some difficulty (there are problems with deleted records and the end-of-file). SUPRSORT supports NOBUF access to KSAM files, while SORT/3000 does not. In a sort of 12,444 records (-80, 16, F, ASCII; KEY=B,40,20), SUPRSORT took 1.7 minutes and SORT/3000 took 5.6 minutes, an improvement of 3.3 times. Since SORT/3000 uses the default access to KSAM files, KSAM must use the key file to put the data records into primary key sequence, after which SORT/3000 sorts them again, into the final desired order. SUPRSORT reads the records in chronological sequence, instead of in primary key sequence. For a KSAM file that has been loaded randomly over time, primary key sequence can require a great deal of disc head movement to retrieve the records.

NOBUF access can also be applied to IMAGE datasets, but SUPRSORT and ADAGER (Adapter/Manager for IMAGE [33]) are the only software tools that I know of which do this. SUPRSORT, for example, retrieves records from an IMAGE dataset by reading the data blocks directly from the disc, rather than calling the DBGET intrinsic for each record. This results in a dramatic savings of CPU time (and elapsed time).

In 1980, I added a generalized selection capability to SUPRSORT (i.e., >IF AMOUNT>100000 AND ORDSTAT<>"X" OR DATE>801030) and changed the name of the software product to SUPRTOOL. In one comparison test, I used SUPRTOOL to select 1033 records from a dataset with 105,504 current entries. When the dataset had 256-word blocks, SUPRTOOL took 222 seconds to do this task, 2.9 times faster than the best SPL program (using DBGET) and 4.8 times faster than QUERY/3000. The effective data rate in this experiment was 11.62 seconds per 1000 sectors of data scanned, versus 34 seconds with the best SPL program. This gain is strictly due to savings in CPU time, since both cases read a single data block per disc revolution.

When the database was reloaded with 512-word blocks, SUPRTOOL took only 110 seconds (twice as fast) and was 4.8 times faster than the SPL program (7.4 times faster than QUERY/3000). The larger block size improved SUPRTOOL's data rate from 11 seconds per 1000 sectors to 6 seconds. (Warning: before you reload your databases with larger blocksizes, read the next section! With the IMAGE database, you can enjoy the benefits of large disc blocks, while still retaining small blocks for your on-line users.)

B3. USE NOBUF ACCESS AND TRANSFER SEVERAL BLOCKS AT ONCE

When you open a file with NOBUF, MPE allows you to transfer one OR MORE blocks on each call. If NOBUF with one block per transfer is good (see previous section), would several contiguous blocks per transfer be even better? Sometimes it is, and sometimes it isn't. I ran a test with SUPRTOOL to demonstrate this phenomenon. From an IMAGE dataset with 251-word blocks (117 words per record, two records per block, 15,255 entries in the dataset), I extracted all of the entries that contained a customer number starting with "X". SUPRTOOL read one disc block at a time and the task took 176 seconds (65 CPU seconds). When I forced SUPRTOOL to read 16 blocks at a time (using DEBUG), the task took 1.44 times LONGER (189 seconds), but used less CPU time (only 45 seconds)!

How can 16 blocks per transfer be worse than one block per transfer? The CPU time was reduced, as we would expect, because there were 16 times fewer calls to the FREAD intrinsic; but the elapsed time increased by 1.44 times instead of decreasing. The answer lies in the BLOCK SIZE: 251 words. Since each block MUST START ON A SECTOR BOUNDARY, a 251-word block must have 128 data words in the first sector and 123 data words in the second sector. That leaves 5 WASTED WORDS at the end of each block. When SUPRTOOL asked MPE to read 16 contiguous blocks from the disc, MPE had to remove those 5 extra words at the end of each block. Since the disc controller is not smart (or fast) enough to perform this chore, the only way that MPE can remove the words is to issue a separate read request for every. Thus, a 16-block read takes at least 15 full revolutions of the disc to complete. When SUPRTOOL asked for one block at a time, each read request took about one-half a revolution to complete (depending upon where the head was when the read occurred). When there are unused words at the end of a block, reading one block at a time is the fastest method.

However, when there are no wasted words at the end of each block, multi-block transfers allow serial processing at rates which reach the theoretical limits of the HP 3000. For example, in copying a file of 32 word records, with 4 records per block, FCOPY has a "copy" speed of 57 seconds per 1000 sectors. SUPRTOOL, using transfers of 4096 words, has a measured copy speed of 1.6 seconds per 1000 sectors, faster than the hypothetical limit of 2 seconds. Of the total 1.6 seconds, 1.46 seconds are spent in either the FREAD or FWRITE intrinsics. The only way that SUPRTOOL can be faster than the "speed limit", is if the actual latency per transfer is less than the average (11.1 ms., or 1/2 revolution). See Section II for the detailed calculations.

Prior to the Athena release of IMAGE (2011), databases were always created with "odd" block sizes, such as 251 words. But, if you build (or rebuild and reload) a database under the 2011 release (or later), IMAGE will round up the block size of each dataset to the next sector boundary (128 words). This was done to allow the DBUTIL program to use multi-block writes to erase a database.

What are the results if I reload my database under 2011 IMAGE and ask SUPRTOOL to scan the database using varying numbers of blocks at a time? In the previous section, I reported an extract job that took 222 seconds to perform, reading one 256-word block at a time from the dataset. Here are the times for the same job, reading 1 to 32 blocks at a time (block size is exactly 256 words):

Number of Blocks:	1	2	4	8	16	32
Words per Read:	256	512	1024	2048	4096	8192
Elapsed Time (s):	223	116	115	89	75	69
Times Faster:	---	1.9	1.9	2.5	3.0	3.2

These times are more like it -- up to 3 times faster. The conclusion that you should reach is: ALWAYS PICK A COMBINATION OF RECORDS IZE AND BLOCKING FACTOR WHICH PRODUCES A BLOCK SIZE EVENLY DIVISIBLE BY 128 WORDS [11], even if you must "waste" a few words per record.

If you follow this rule, you can achieve data processing speeds that are close to the theoretical hardware limits (1 second per 1000 sectors). SUPRTOOL, for example, performs selective data extracts on "even" block sizes at rates up to 3.6 seconds per 1000 sectors. Straight copy operations, without selection, are even faster: 1.6 seconds per 1000 sectors. When doing selective extracts, SUPRTOOL spends .75 seconds per 1000 sectors in the FREAD intrinsic. That is faster than our target speed limit. It shows, once again, that CPU time has the power to stretch out the elapsed time for any task. SUPRTOOL uses 2.85 seconds per 1000 sectors to evaluate which records to select (and to move them to output buffers) and only .75 seconds per 1000 sectors to read and write the data blocks.

Another advantage of using NOBUF reads of the database (as SUPRTOOL does), instead of calling DBGET, is that a serial search does not disturb the shared "buffer pool" in the global DBCB extra data segment (see topic A3 above). If there are other users accessing the database, a batch program that makes 100,000 calls to DBGET in a short time will "flush" the buffers that are allocated to other users. NOBUF access bypasses these buffers completely.

With an "even" block size, the speed of multi-block NOBUF is exactly as fast as single-block NOBUF using very large blocks. Therefore, you can keep your actual block sizes small (but they must be divisible by 128 words with no remainder) to optimize on-line access, but not too small (I suggest 512 to 1024 words) to optimize cases where you choose to use buffered access instead of NOBUF access. Since IMAGE now rounds up blocks for you automatically, I suggest that you use the default IMAGE block size (512 words), except in cases where a record size does not block well into 512 words (see A3 and A6 above). If you have an existing database with "odd" block sizes, you can convert to "even" blocks either by building a new database and reloading from tape (DBLOAD), or by running the new REBLOCK function of ADAGER/3000. For magnetic tape files, I suggest a very large block size (4000 to

8000 words) and NOBUF access, one block at a time. Multi-block tape transfers seem to crash some releases of MPE.

B4. USE DBCONTROL TO DEFER DISC WRITES BY IMAGE

Normally, after the completion of each IMAGE intrinsic call, the IMAGE database system flushes all "dirty" blocks in the buffer pool back to the disc. This is one of the reasons IMAGE data structures are so reliable: the time window during which they are inconsistent is very small.

The DBCONTROL intrinsic provides a way of disabling the automatic posting of dirty blocks. It has been suggested that DBCONTROL be used to improve large batch jobs [34]. The DBLOAD utility, for example, uses this feature to accelerate the loading of databases from tape. However, if the system should crash while you are in "deferred-output" mode, you must RESTORE your database from a backup tape. Otherwise, there could be serious logical and structural inconsistencies in your database.

One user test of DBCONTROL found a reduction in elapsed time from 21 minutes to 16 minutes for one application, and from 22 minutes to 8 minutes for another [04]. These are improvements of 1.3 times, and 2.75 times, respectively. It is likely that tasks requiring a large number of puts and deletes (relative to gets and updates) will show the biggest improvement. Designers are often encouraged to avoid doing on-line deletes from the database by flagging "dead" records and deleting them with a batch program [45]. Such a batch maintenance program is a good candidate for DBCONTROL, since it can easily be scheduled to run after a database backup (essential to avoid losing your database without any way to recover).

When I upgraded my SUPRSORT utility into SUPRTOOL, I added the ability to write the output records to an IMAGE dataset (i.e., do a DBPUT). I also included a command to enable "deferred-output" mode through DBCONTROL. Using these features, I loaded 2551 records into a detail dataset that had a single key field. In normal mode ("complete-output"), this task took 333 seconds; in deferred mode, it took 156 seconds, an improvement of 2.13 times. (See topic A3 above for related ideas.)

The dramatic difference between a DBPUT and an FWRITE to a sequential file can be seen by calculating a "data rate" for DBPUT. In the test mentioned above, with a single path to update, DBPUT had a rate of 203 seconds per 1000 sectors of data in "complete-output" mode (default) and 94 seconds per 1000 sectors in "deferred-output" mode. This compares with a rate of 1 to 2 seconds per 1000 sectors for NOBUF, multi-block transfers using FREAD and FWRITE. The reason DBPUT takes so much longer is that it must do random (i.e., "hashed") reads from the master dataset (about 1570 reads per 1000 sectors, plus 1570 writes, times .03 seconds per transfer, equals 94 seconds).

B5. RECOMPILE COBOL PROGRAMS WITH COBOL II COMPILER

Hewlett-Packard has run a large number of tests to measure the performance of COBOL II, relative to COBOL/3000 (also known as COBOL I or COBOL 68). In analyzing their results [22], I have noticed a number of interesting points. Batch jobs should run a bit faster (1.1 to 3.0 times faster), depending upon how much I/O they do (compute-bound programs are improved the most). On-line programs use less CPU time, but users do not see any improvement in response time (since on-line programs spend most of their time in IMAGE and V/3000).

The big surprise is that data stack sizes are often reduced by 25-60% due to the elimination of RUNNING-PICTURE and PARAGRAPH-RETURN tables. For on-line users, this "should" mean you can run more terminals with the same memory (unless you are already disc-bound). For batch programs, this savings in stack size can be used to eliminate disc transfers by copying tables (i.e., small master datasets) from the disc into the stack. The only problem with COBOL II is getting a version that works right. The "Cheetah" version appears to have most of the bugs worked out (HP32233X.00.02C?).

B6. REWRITE SOME CODE IN SPL/3000

Rewriting COBOL programs in SPL is a technique that I have recommended for several years [17]. However, this does not apply to batch programs. Without proper training materials and programming guidelines [16], you could actually make things slower by rewriting COBOL code in SPL. I can give two examples that I have seen in actual user sites.

First, by converting to SPL, you lose the built-in data conversion power of the COBOL language. You you must code explicit conversions between ASCII and BINARY formats. If you use the intrinsics provided with MPE to perform this task, your batch tasks will take longer in SPL than in COBOL. The MPE intrinsics take 3 times longer to execute than substitute routines that I wrote in SPL. This is probably due to the enabling and disabling of the error trap mechainisms which must be done in every MPE intrinsic.

Second, since SPL has no "packed decimal" data type, you may be forced to write (or acquire) SPL subroutines to do packed-addition, packed-subtraction, etc. Because each addition (subtraction, etc.) now requires a procedure call, these tasks will take significantly longer than they do in COBOL (which generates in-line code). These considerations are even more applicable to COBOL II, because it consumes less CPU time than COBOL 68.

In summary, while it makes sense to rewrite MPE intrinsics (such as ASCII) in SPL and it may make sense to rewrite high-usage COBOL subroutines in SPL (i.e., checkdigit, edit validations), it does not make sense to rewrite batch reports in SPL. SPL should be reserved for on-line optimization, for specialized support routines (access to MPE capabilities and hardware features), compute-bound subroutines which are called frequently, and utility software. Many of the optimizing techniques mentioned in this paper be used most easily in SPL, but you do not need to hire SPL programmers to take advantage of them. You can buy the optimizing tools, already coded, debugged and documented, from a software vendor. (If you want to develop SPL expertise, see [16] for suggestions.) Production batch jobs should be written in COBOL II (or RPG or FORTRAN), unless there is some compelling reason to write them in SPL.

B7. BYPASS THE FORTRAN FORMATTER (ETC.)

Each implementation of a programming language has problems. In FORTRAN/3000, the notorious trouble spot is the "formatter" (processing of FORMAT statements) [43] [40]. In a bizarre example sent to me by a customer [43], performance of a particular program varied by a factor of 58 times, depending upon how the READ statements were coded. The differences were totally a matter of CPU time spent in the formatter (the program read 4600 records, 2040 bytes each, blocked 4, using default buffering).

```
Case 1.  4632.8 CPU seconds (over 1 hour to read 4600 records!)
               3  FORMAT ( 2040 ( A1 ) )
                  CHARACTER A2040*1(2040)
                  READ(10,3,END=2) A2040
Case 2.  108.91 CPU seconds (42.5 times faster)
               3  FORMAT ( 10 ( A204 ) )
                  CHARACTER A2040*204(10)
                  READ(10,3,END=2) A2040
Case 3.  79.16 CPU seconds (58.5 times faster, without format)
                  CHARACTER A2040*1(2040)
                  READ(10,END=2) A2040
```

Apparently, the compiler generates 2040 calls to the formatter for each record in Case 1. Horror stories like this prove that CPU time matters. In batch processing, a small inefficiency in the innermost loop can turn into a big increase in elapsed time, when it is repeated a million times (as above). The programmer should learn the constructs to avoid in his language, whether it is FORTRAN, COBOL [22] or RPG [46].

B8. ELIMINATE UNNEEDED DATA CONVERSIONS

One area where standard programming languages usually have performance pitfalls is in data types. Standard languages must "map" their logical data types into the available hardware data types of the HP 3000. When the types match closely, performance is good. When they do not match, the compilers must generate "fixup" code and performance can be bad. In COBOL, for example, numeric data fields can be either signed (S9(4)) or unsigned (9(4)), but most of the HP 3000 hardware data types are only signed. Therefore, according to Hewlett-Packard [22], "using signed instead of unsigned data avoids the need for computing the absolute value of a result after it is obtained. This affects COMP-3 and DISPLAY items more than COMP, and can result in a moderate savings in execution time."

Many implicit data conversions occur in standard programming languages, especially (but not exclusively) when you mix data types in expressions. In COBOL 68, it is rumored that COBOL, all COMPUTE statements are done using packed-decimal arithmetic, regardless of the original data types. In a benchmark that I read about, a COBOL 68 program was reduced from 10 minutes to 12 seconds by changing "ADD 5 TO SUM" to "ADD FIVE TO SUM", with "FIVE" defined in the data division as a numeric field with the same data type as SUM.

In FORTRAN, mixed-mode expressions are discouraged by S.E.'s [31] [40], because conversions occur. In RPG, users are encouraged to avoid using numeric display or binary field types, since RPG must convert them to packed-decimal to perform arithmetic [46].

Saving a few milliseconds may seem like a minor matter. For ON-LINE programs, it IS minor, because on-line programs seldom use enough CPU time in an entire day to be justify optimizing them. But, if something is repeated often enough, the inefficiencies in CPU time start to add up to a significant amount of elapsed time. That is why NOBUF (in place of MPE buffering or the interface of DBGET) makes batch jobs run faster.

In summary, the potential exists in many applications for major improvements through more detailed knowledge of the machine-dependent features of programming languages. However, I strongly suggest that you start the habit of verifying each suggestion before implementing it. This is not difficult to do. Start writing test-case programs, with subroutines to measure the CPU time consumed. In one example that I saw recently [04], the user had followed advice to convert data items to "binary" (from what?). Instead of running faster, the elapsed time increased from 21 minutes to 31 minutes. You should not put total faith in everything you read. Compilers change. MPE changes. Hardware changes. Keeping up with these changes is a continuing process.

B9. REDUCE NUMBER OF "OPENS"

I once optimized a program that took an entire weekend to prepare and print monthly bills for 18,000 customers. The user client needed the program fast enough to produce 400,000 bills per month. One of the most blatant problems in the program was the result of last minute specification changes. An "audit" database was added to the application. The transaction subprogram was modified to create an audit copy of each transaction. To do so, the programmer opened the audit database, did a DBPUT, then closed the database. This meant a minimum of 18,000 extra calls to DBOPEN and DBCLOSE. I changed the program to open the audit database once in the mainline, and to pass in the database name to the transaction subprogram. This one change cut the elapsed time by 15 hours.

Sloppiness of this kind often creeps into batch programs when changes are made at the last minute, and schedule dates are not extended. Testing seldom catches programming errors of this kind, because the the volume of test transactions is usually too small. Other operations that should not be performed 18,000 times are FOPEN, RENAME and SORT.

As I said in my earlier paper [15], one method of optimizing is to focus on the common events. Start with the tasks that are repeated the most times. Ascertain that they are efficient, before worrying about tasks that only occur one tenth as often. You can use the MPE log statistics to detect files which are heavily accessed (or opened too often) [42], and to select programs for review which run often (or for too long).

B10. INCREASE SIZE OF CODE/DATA SEGMENTS

In reading about optimizing, you will see suggestions on segmenting programs into small (4K word) code segments. Code segmentation reduces the impact of a program on the rest of the system, but it DOES NOT MAKE THE PROGRAM RUN FASTER. Since a "local PCAL" (calling a routine in the same code segment) takes 6.1 microseconds, and an "external PCAL" takes at least 14.9 microseconds (34 milliseconds, or more, if the code segment desired is not present in memory), segmenting your program actually makes it run slower. Stand-alone batch programs may run slightly faster with fewer, but larger, code segments. (You should still put routines that call each other in the same segment). I tested this hypothesis by calling data conversion routines 64,000 times. When the routines resided in the same code segment as the calls, the execution time was 1.06 times faster than when the routines were in a separate code segment.

My general point is this: avoid being trapped into inflexible thinking by the constant emphasis upon on-line programming in most manuals and optimizing documents.

Another common prescription for bad performance is to shrink your data stack using the ZSIZE intrinsic [15] [39] [22] [18] [32]. If your stack is larger than necessary for long periods of time, it impacts the response time of other on-line users. It takes only a few milliseconds to contract your stack to give back the unneeded space. But, it takes considerable time to expand your stack again (which is done automatically when you need the space). Stack expansion requires a disc write and a disc read [23]. In a batch program, if you contract your stack after each transaction, you may have a very slow program, if MPE expands your stack again for the next transaction.

When an on-line application program is coded to shrink the stack dynamically, the savings are multiplied by the number of users running the program (1, 10, or 30). Big improvements in response time are possible through this technique because large amounts of main memory are freed for other users. Now many copies of a batch job run at once? Usually only one copy. Reducing the size of your batch stack does not increase the speed of your batch program; it only contributes marginally to the response time of other jobs. Given the rapid increase in main memory capacity of the HP 3000, I suggest that batch stacks be increased whenever that will lead to a decrease in CPU time or disc accesses.

For example, consider the practice followed by many HP 3000 programmers of eliminating global data storage by making it local. Instead of making data "dynamic", consider making more of it "static" (global, common, own, subprogram). Allocating local storage "dynamically" each time you enter a subprogram conserves stack space, but it takes CPU time to do the allocation.

The ideas in this section are controversial, and, until sufficient tests have been performed to verify their impact on throughput, they should be treated as "promising", but unverified, proposals. Users are encouraged to do their own performance measurements in this area.

C. CHANGES TO APPLICATION DESIGN

When all possible improvements have been made to data storage methods and programing efficiency, the only area left for attention is the logic of the system. Although design changes are the most expensive to implement, they also provide the greatest potential for gain. If you can eliminate the need to run a certain report, your percentage gain in throughput is infinite.

C1. USE DATA STACK SPACE INSTEAD OF DISC ACCESS

In the previous discussion (topic B10), I suggested that you look for ways to trade off a larger data stack for fewer disc accesses. For example, if a program uses a small temporary file as a buffer space, the entire "file" could be kept in the stack as a large array. (This will only improve performance if the file is accessed many times per transaction.) One of my clients has an invoicing program that uses a temporary disc file to hold the line items of each order (after they are copied in from the database). The number of line items per order varies from 1 to 50, with an average of 3. According to the contributed program FILERPT [42], this file is the second most heavily accessed file on the system. If I add subroutines to the program to simulate the temporary file using an array in the stack, I should be detect a significant decrease in elapsed time for a given invoice run.

Another candidate for a place in the data stack is a "lookup table" for validating data fields. In many batch processing tasks, each transaction record has one or more fields that must be checked against a list of valid values in an IMAGE master dataset. If the number of valid values is small enough, you can move the entire master dataset into your stack (4000 to 16,000 words). This strategy would not make sense for an on-line program, because there are too many copies of the stack that must reside in memory at the same time. Also, on-line programs seldom do enough table lookups per minute to justify each user having a copy of the entire dataset in memory. For a batch task, however, there is only one copy of the stack, and there may easily be 100,000 table lookups in one run.

How much time could the in-stack table save? A computed DBGET takes about 75 milliseconds (including one or two disc reads). For a batch job to do 100,000 DBGETs, it takes about 7500 seconds, or 125 minutes, or 2 hours. Most of this two hours would probably be eliminated by an in-stack table (how much depends upon how long it takes to search the table in the stack).

C2. USE EXTRA DATA SEGMENTS INSTEAD OF DISC ACCESS

When you have exhausted the 64,000 bytes that are possible in your data stack (copying tables and temporary files into the stack, as proposed above in topic C1), there is one more resource that you can tap before you use a disc file: Extra Data Segments (XDS). An XDS is a "chunk" of data space that belongs to your program, but is swapped in and out of memory by MPE, independently of your data stack. Your program accesses data within the XDS via the DMOVIN and DMOVOUT intrinsics. The size of an XDS can vary from a few words (why bother?) to 64,000 bytes (the maximum size is a system configuration value). Each program can create and access up to 255 XDS (also limited by system configuration).

The HP 3000 has a lot of main memory to work with, so why not use it all for a stand-alone batch job? If you have one megabyte of real memory, of which MPE uses 128k and your program needs 256k (for stack, IMAGE DBCB segments, code segments, etc.), you still have 655,360 bytes left. If you limit your XDS size to 16,000 bytes (8000 words, an easy size for MPE to swap), you can have 40 XDS and still not be swapped (this is all theory, subject to experimental verification).

According to one article, an XDS is 25% to 900% faster than a disc file [24]. Of course, it is not as fast as data in your stack, so use the stack for the most frequently accessed data. Access to a subscripted variable in COBOL takes about 60 microseconds, while access to an extra data segment takes 1.3 milliseconds [25]. Access to a buffered MPE file takes 2.9 ms. if the record you want is in the buffers, and 40 ms. (or more) if a disc transfer must be done.

In 1980, David Greer of my staff did a research project to investigate the substitution of main memory resources for disc access resources [19]: "One answer is a memory file. This is a file that looks, and is accessed, as if it were on the disc, but which actually exists in memory. The savings could be great if the number of disc accesses is very high, but the cost should be low, since systems are all tending towards more memory as memory costs continue to drop. The memory file should be implemented with little or no knowledge of the applications programmer. One way to do this would be with a special MPE device class (such as MEMORY). The first time a file with this class is opened, it would be read into main memory. All accesses to the file from then on would be memory to memory, rather than disc to memory. When the file is closed, it would be copied back to the disc. A prototype memory file system was implemented (for exclusive-access files only), using a number of extra data segments, including one to hold control information. All file system calls from the application program were intercepted by interface routines that emulated the MPE file system. Test programs showed that the MPE file system was faster than this double-XDS system UNTIL the MPE file buffers were exhausted (default = two buffers). After that point, the average times for MPE files increased 4 times, while the memory file system times remained steady. The results show that a memory file system could provide a major improvement over the regular file system, for small files that are heavily accessed."

The contributed library contains TBPROC [37], a set of SPL routines that allows a COBOL program to maintain a table in an XDS (routines are provided to define a table, add entries to a table, update entries, sort the table and retrieve entries from the table by key value or relative index). TBPROC allows you to define the maximum number of table entries, the byte length of each entry, the offset of the key value and the byte length of the key value. The author of TBPROC wrote it to hold small IMAGE master datasets, as proposed above. He feels that this change reduced the elapsed time of a large production job by one-third [36], but cannot be certain (since other factors were changed at the same time). I would like to enhance TBPROC to allow for an optional buffer in the stack. The user-supplied buffer would be used instead of the XDS, when the table was small enough. If the table overflowed this space, it would be stored in an XDS and the in-stack srace would be used to optimize access to the XDS.

Another contribution to the library, ARHND, uses up to 64 XDS of 16,000 bytes each to provide one megabyte of virtual array space for a program. Where TBPROC is COBOL-oriented and provides table lookup by key-value, ARHND is FORTRAN-oriented and provides up to 13 virtual arrays (of single or double integers). The size of the arrays can be varied dynamically, and they can be one or two dimensions. Routines are provided to allow the program to emulate sequential files using a virtual array. Using ARHND, it appears that a single FORTRAN program could consume the entire resources of an HP 3000.

There is a certain unavoidable overhead in the DMOVIN and DMOVOUT intrinsics, because they are part of the MPE operating system (if MPE had a NOOP intrinsic, it would take at least ONE millisecond). For maximum speed of access to an XDS, you can use the hardware instructions that move between data segments [14]. Of course, this requires privileged mode and careful programming on your part. Another option is to use EXCHANGEDB and operate in "split-stack" mode. This also requires privileged mode. The

BASIC-callable version of the "BREAD" NOBUF routines uses "split-stack" [13], because the BASIC Interpreter claims total control over data space allocated in the stack. I mention these options only for completeness; I have never seen them put to use in an actual, commercial batch program. They might be very useful, however, in software products (such as [13]) which are designed to support and optimize production batch programs.

One thing about XDS usage worries me. An XDS can be swapped out by MPE if the memory space is required for a higher priority process. When the program next references the XDS, MPE must swap it back into memory again. It might be faster to use NOBUF disc files and manage buffers in your stack (assigning buffers dynamically to the "active" blocks; that way, you control the level of swapping that occurs.

When Hewlett-Packard implemented APL on the HP 3000, they needed a large virtual data area for the APL workspace. Rather than use XDS, they created a new data structure called a "Virtual Array". A VIRTUAL ARRAY can be declared in SPL, but requires the APL firmware at run-time. The APL virtual array is stored on the disc as a series of fixed-length "pages", and the APL interpreter allocates a certain amount of in-stack space as a buffer pool (to hold some of the pages). When the interpeter references a virtual array, the special firmware instructions check to see if the required page is in the buffer pool. If it is, the virtual data is accessed immediately. If it is not, the "least recently used" page is swapped out and the page containing the desired data is swapped in.

I would like to see a controlled experiment matching the ARHND-type of virtual array against a stack-NOBUF-file type of virtual array. This entire area has tremendous potential for optimization of batch programs, but needs a great deal more research.

C3. SAVE DATA OR POINTER FOR RE-USE ONCE RETRIEVED

Having retrieved a specific record(s) from a detail dataset (using 5-10 IMAGE calls), you should save the record number (found in the STATUS array) [45]. It can be used for a DIRECTED-GET later, when you are ready to update or delete the record. Does each subprogram of your COBOL program re-retrieve the customer-master record from the database when it needs it? You should only retrieve each customer-master record once, store it in a global data division, and pass it to the subprograms as a parameter? Does your program validate transaction fields by doing lookups in the database? You should save the key value that was tested for the last transaction in a global place, and check there before checking the database (or keep a working set). IMAGE adds entries to unsorted detail-dataset chains in chronological order. Therefore, if the entries that you need are more likely to be recent than old, you should read up the chain (mode 6), instead of down the chain (mode 5).

A program spends most of its time doing "work" to convert dispersed, raw data into accessible, organized information. Once the program has converted the data into accessible and/or organized information, the program should save that information, if there is any chance that the same information will be needed again soon. Otherwise, it must repeat the work later.

C4. REPLACE MULTIPLE PUTS/DELETES WITH UPDATES

DBPUT and DBDELETE take 5 to 20 times longer per call than DBUPDATE, because DBUPDATE is not allowed to change "critical" fields (search items or sort items). That is, it cannot make "structural" changes to the data. For each path path into or out of a dataset, DBPUT and DBDELETE must update the chains that link entries with the same key value (in a detail dataset) and the chain heads (in the master datasets). This task requires disc accesses and CPU time. The more paths there are into an entry, the longer it takes. That is why optimizing papers often recommend that on-line programs merely flag "dead" records and let them be deleted in batch [45]. Similarly, if you are not changing any critical fields, you should always use DBUPDATE instead of deleting the entry and adding it again [34].

One way to replace PUT/DELETE with UPDATE is to eliminate paths (see topic C5 below). Another way is to merge several independent entries (that must be PUT and DELETEd) into a single entry, with a "column" for each individual piece of data. For example, instead of creating an entry for each month of the year (12 PUTs and 12 DELETEs per year), use one entry for the entire year (1 PUT, 1 DELETE), and UPDATE the entry when you need to record the value for a particular month (12 UPDATEs).

I tried this approach with one client's application. The previous consultant had designed a detail dataset to hold billing transactions, indexed by customer number and sorted by date. Customers were normally billed once a month, for a recurring fixed amount (i.e., $5.00 per month). When they paid, another detail entry was created to show the payment. In order to examine the status of an account, it was necessary to retrieve all of the entries on the chain (24 entries minimum for a year). But, the transactions were predictable (billed $5, paid $5). We replaced the individual transactions with a single entry that showed the amount to be billed per month, and had places to be "checked off" when each month was billed and then paid. This replaced 24 PUTs and 24 DELETEs (you have to get rid of those transactions someday), with 1 PUT and 24 UPDATEs. The impact on batch throughput (as well as on-line response) was impressive; and, this approach was actually closer to the way the company had kept track of its customers before the computer was installed.

C5. CONSIDER SERIAL SEARCH INSTEAD OF CHAINED ACCESS

On-line functions should only do chained (or keyed) reads from the database. They should never do serial scans of an entire dataset (unless the dataset is very small). This is what search items are for: to provide quick response to inquiries.

For a batch program, "response time" (i.e., the time it takes to retrieve a subset of entries from a dataset) does not have to be immediate, it only has to be reasonable. An application may actually run faster (overall) if you eliminate a search item that is accessed exclusively in batch, and use a serial scan instead [45]. The time to PUT and DELETE those records will be reduced, and you may be able to do an UPDATE instead of a DELETE/PUT (if the field to be updated is the deleted path or the sort item). Some of the search items that I have eliminated from datasets are "division number" (where there are only four divisions), "transaction date", "transaction month", and "salesman" (with only 10 salesmen).

In fact, if a given path has only a few unique chains and each chain is very long (i.e., only a few values for that field), a serial scan MAY ACTUALLY BE FASTER than a chained read. This inversion of logic is most likely to occur in datasets that have had many DELETEs and PUTs since the last DBLOAD reorganization. Such activity tends to spread the entries that are on the same "logical" chain into different physical disc blocks (because IMAGE reuses deleted space for new entries). Each chained read can, therefore, require a separate disc read. Serial reads, on the other hand, only do a disc read once for each N entries (N=blockfactor).

In one test that I ran, chained reads took 16 milliseconds each, and serial reads (using DBGET with *) took only 5 milliseconds each. In this case, the chained retrieval will only be faster if there are four or more unique key values (each chain has less than one quarter of the entries). With SUPRTOOL, the time per serial read is much faster, only .63 milliseconds. As a result, SUPRTOOL will be faster than chained access for any path that has less than 25 unique chains.

C6. ISOLATE DATA BY FREQUENCY OF ACCESS

Do you plow through the transactions for an entire year every night, in order to produce an audit report of the activity for the day? Why not put the day's work into a separate dataset? After producing the audit report for the day (which should finish 30 to 400 times faster) and re-validating the transactions (to catch bugs in the on-line programs), move the entries into a month-to-date dataset and delete them from the daily dataset. After month-end closeoff, move the month-to-date entries to the year-to-date dataset. Another benefit of this approach is that each dataset can have a different type of access (different number of search items and sort fields). Finally, after the end of the fiscal year, you can copy the year-to-date dataset to a disc file (never throw away good transactions) and clear out the dataset for the next year. The archive disc file can easily be reported from, since it has the same format as the year-to-date dataset. It can also be stored to tape if there is no immediate call for it. (SUPRTOOL has commands to perform most of these extract/copy operations, and with excellent speed.)

In the previous discussion (topic C5), I suggested eliminating search items that have less than 4 unique values (less than 25, if you have SUPRTOOL). Now I am suggesting that you create separate datasets to isolate entries, when the distinguishing field has only 3 to 5 active values (such as range of date = current day, current month, current year, or other year). "Isolation" reduces the number of physical entries that your serial search programs must read. They need only look in the relevant datasets.

C7. KEEP RUNNING TOTALS IN DATABASE - ELIMINATE SEARCH

I have seen many batch applications which must re-scan all of the transactions for the year (or two years), in order to compute sub-totals by account number, month, division, etc. Once each month, they sort all of these transactions in two or three different ways. Several users have reported to me job times of 20 hours or more. With the availability of the IMAGE database, there should be few designs of this kind. Once a total has been calculated, and you know that you will need it again next month, it should be stored in the database for quick retrieval. The basic principle that I try to follow in disposing of information is: reduce transactions to summary totals as soon as possible, while saving the transactions (with their wealth of detail), in case a question comes up that cannot be answered from the summary information.

General ledger packages seem to be the worst offenders of this rule -- especially the ones that were converted to the HP 3000 from card-oriented IBM systems. By far the most elegant use of IMAGE for a general ledger that I have seen is the hierarchical structure (trees of accounts with sub-totals at each node) that is described in [32]. Anyone designing a new general ledger system should read that paper for ideas.

By saving summary totals, you are "depositing" into the database the CPU time and disc accesses that were used to calculate them. When you need those totals the next time, you have only to look in your safe deposit box (instead of taking out a loan to "buy" the information for a second time).

C8. SORT DATA BEFORE WRITING TO KSAM FILE

Within IMAGE and KSAM, there are "sorted" data structures -- sorted chains for IMAGE and sorted keys for KSAM. The time needed to add entries to such a structure may be reduced, if the entries are in sorted sequence initially [45]. (Of course, if there are multiple sorted keys, only one key can be optimized.) I used

SUPRTOOL to test this hypothesis on KSAM. When SUPRTOOL copied 2551 records to an empty KSAM file (with one key, duplicates allowed), the elapsed time was 92 seconds. KSAMUTIL reported that 2224 key block I/O transfers had been required. SUPRTOOL then repeated the operation, but sorted the 2551 records before writing them. The total time was reduced to 46 seconds, including the sort time (2 times faster), and the number of key block I/O transfers was reduced to 134 (17 times less).

In the tests that I performed, KSAM obtained a load "data rate" of 111 seconds per 1000 sectors with unsorted data and 53 seconds per 1000 sectors with sorted data. Compare this with the rates for DBPUT (203 seconds in default mode and 94 seconds in "deferred-output" mode, which is the way KSAM operates) and the rates for NOBUF calls to the file system (1 to 2 seconds per 1000 sectors). The more organization you demand of your data, the lower the rate at which it can be updated.

C9. SORT ENTRIES BEFORE PUT TO SORTED CHAIN

Since sorting data before writing it to a KSAM file cut the elapsed time in half, I hoped that the same thing would be true for IMAGE. There are two major differences, however, between KSAM and IMAGE. First, the entire KSAM file is sorted by the key field, while only a single chain is sorted in IMAGE. If the IMAGE chains are short (average length less than the blocking factor), it may not take IMAGE very long to put them in order. Second, KSAM always operates in output-deferred mode (it does not post buffers at the end of each FWRITE). IMAGE operates in output-complete mode. Therefore, the disc transfers needed to update the sort sequence will probably be a larger percentage of the total disc transfers in KSAM than they are under IMAGE. I have not seen a satisfactory experiment with these ideas yet.

If you do try this technique, I suggest that you either limit it to large batch tasks (where you sort an entire dataset before copying it to another dataset), or that you write your own internal sort routines. There could be nothing more catastrophic for your throughput than to do 10,000 seperate sorts, one for each sorted chain. Each time that you initiate the Hewlett-Packard sort sub-system (whether by running SORT.PUB.SYS, or using the SORT VERB in COBOL, or calling SORTINIT in SPL and FORTRAN), you are causing a temporary file with 10,000 records to be allocated on the disc. If you are only sorting 15 to 100 records, this is a tremendous waste of resources (both disc accesses and CPU time).

C10. COMBINE KSAM WITH IMAGE IF YOU NEED SORTED ACCESS

If sorted chains in IMAGE are bad [45] [34], and invoking the Hewlett-Packard sort package for each chain of 50 records is also bad (too much overhead to initiate and terminate the sort), what else can be done? Sometimes you need sorted access to IMAGE entries. Sorted access is the easiest way to provide generic search capability. IMAGE is not always the best answer. Sometimes you should use KSAM.

Here is a suggestion: copy your key values from an IMAGE master dataset and write them to a KSAM file (single key, no duplicates) in sorted order. One of the references that I read [38] found that sorting the key only (instead of the full record) could save up to 41% of the sort time. My tests (reported above, C8), found that sorting records before writing them to a KSAM file would cut the load time in half. Therefore, this IMAGE-to-KSAM transfer should be very quick. This is one of the tasks that SUPRTOOL can accomplish (extract and sort keys, write them to a KSAM file, clearing it first).

If you follow this prescription, you will have an "updatable" mechanism for sorted access to an IMAGE dataset. If the key field is "date", and you want to find all entries with dates in the month of June, you do a FREADBYKEY into the KSAM file and use the key values retrieved to get the actual records from the IMAGE dataset. If you want your index updated during the day, you must modify your on-line programs to FWRITE new key values to the KSAM file. Or, you may be willing to have the sorted access updated only once a day (after new transactions are validated in batch). In this case, just run the SUPRTOOL copy operation once every night.

Section V WHAT "ACTUALLY" MAKES BATCH FASTER?

The last section evaluated many possible techniques to make batch jobs faster; the results are summarized below:

THESE TECHNIQUES WORK (sorted by their "power" to improve)

3x to 300x faster: Isolate data, at design time, by frequency and type of access.
5x to 50x: Save summary totals in a database.
5x-58x: Bypass the FORTRAN Formatter.
5x-20x: Eliminate unneeded data conversions.
10x: Convert DBDELETE/DBPUT to DBUPDATE.
2x-10x: Use NOBUF with MPE files (see FASTIO, BREAD).
2x-10x: Use NOBUF with KSAM/IMAGE (see SUPRTOOL).
3x: Rewrite some MPE routines in SPL (i.e., ASCII/BINARY).
2x-5x: Use more key block buffers, if a KSAM file is empty.
2x: Sort records before loading them into a KSAM file.
1.5x-2.5x: Use DBCONTROL to defer IMAGE writes, if the database is backed up to magnetic tape first.
1.3x-3x: Convert from COBOL 68 to COBOL 74 (COBOL II).
1.1x-3x: Increase IMAGE buffers for a single batch job to the maximum allowed.
1.5x: Keep IMAGE master datasets "clean" (see DBLOADNG).
1.5x: Use "*" field list, with write access to the dataset.
1.2x-1.5x: Increase the block size of MPE files to 1024 words, and that of IMAGE datasets to 512 words.

THESE TECHNIQUES LOOK GOOD, BUT NEED MORE TESTING

Use DBLOAD occasionally to reorganize your database. Reload the entire system to reduce disc fragmentation. Rewrite COBOL subroutines in SPL, if they are called frequently. Increase the size of code and data segments (the opposite of on-line optimizing). Use in-stack tables and extra data segments to eliminate disc accesses. Save pointers and data to eliminate some disc accesses. Consider serial search in place of chained and drop the search item. Sort data before DBPUT to a long sorted chain. Use KSAM with IMAGE to provide sorted access to IMAGE data.

THESE TECHNIQUES DO NOT WORK AS ADVERTISED

Using more than the default MPE file buffers. Dividing program code into small code segments (on-line programs only). Repeatedly contracting the data stack (on-line only). Doing extensive work to optimize head locality through explicit placement of files (except for straight file copies). Increasing IMAGE block sizes above the default (512 words). Increasing MPE file block sizes above 1024 words.

Section VI CONSEQUENCES FOR DIFFERENT PEOPLE

The findings of this report have different implications for the different groups of people associated with an HP 3000 installation.

END-USERS: Ask for summary information and exceptions instead of mountains of paper; you will get your results faster.

DP MANAGERS: The "stock" HP 3000 may have to be "souped up" to handle your batch processing load. Try to acquire optimizing tools such as SUPRTOOL, BREAD and FASTIO, and give your staff the time needed to investigate the contributed library. Establish procedures to measure the resources consumed by each batch program, and review the results quarterly. Question changes to specifications at the "last minute" and insist on adequate time to implement them properly. Invest in staff development (through training and the users group) to keep up with the changes in hardware and software. Don't skimp on disc space. Ask systems analysts to estimate elapsed time for batch jobs, not just response time of on-line programs, when they design an application.

OPERATIONS STAFF: Don't spend hours moving files to specific disc drives. Run DBLOADNG once a month to check master dataset hashing and detail dataset "randomness". Consider DBLOAD occasionally.

APPLICATIONS PROGRAMMERS: Eliminate disc accesses wherever possible, by using DBCONTROL to defer writes in IMAGE, by copying small master datasets into the data stack, by doing multi-block NOBUF transfers, and by using extra data segments to access the main memory that is outside of your data stack. Save CPU time, especially in modules that are invoked in a loop, by avoiding inefficient constructs in your programming language, by switching to COBOL II, by using NOBUF access, by writing subroutines in SPL, by using the "*" field list with write access, and by hand-coding some operations, instead of using the general-purpose software provided by Hewlett-Packard (e.g., small sort operations). Measure the throughput of your programs using alternative methods. Avoid doing the same "work" twice when you could save the results for re-use later. Concentrate your efforts on "common events", such as the functions of a program that are executed the most often.

APPLICATIONS DESIGNERS: Keep record sizes below 256 words and block sizes between 512 and 1024 words, with the block size always a mutliple of 128 words. Design the database to hold summary totals, rather than recalculate them from the original data every time a batch report must be run. Add datasets to isolate records that may have the same fields, but are used with a different frequency and type of access (serial versus chained). Eliminate some search items with few values and use serial scan intead (use SUPRTOOL for maximum speed). Save search items for on-line access. Combine several records into one and use DBUPDATE instead of DBDELETE/DBPUT. Select search items that will hash well (X8 instead of J2). Consider KSAM for sorted access to IMAGE key values. Identify batch processing tasks at design time and estimate their execution times; use these estimates when designing the database. Develop guidelines, goals, and strategies for setting up job control commands, just as you would with any other high-level programming language.

SYSTEMS PROGRAMMERS: For COBOL (and other standard languages), add a built-in deblocking capability and make use of extra data segments where applicable. For IMAGE, do multi-block reads when a serial DBGET is requested and the buffers are not "busy". Improve the job control language of MPE so that jobs can handle more situations without needing operator intervention.

SYSTEMS SOFTWARE DESIGNERS: Provide special "paths" for batch programs so they can avoid slow, general-purpose systems software (IMAGE, Formatter, file system). Concentrate on making full use of the existing disc hardware speed. Merge KSAM capabilities into the IMAGE database.

HARDWARE DESIGNERS: Build a smarter disc controller (on a full track transfer, begin wherever the heads are located, and adjust the memory address, instead of waiting for the disc to rotate to the correct sector). Add CPU instructions to perform deblocking and database processing. Expand the data stack size to at least one megabyte. Provide multiple disc channels. Build a "black box" to do sorts.

Appendix | REFERENCES (SORTED BY AUTHOR)

For HP 3000 users who would like a "self-teaching course" on optimizing, I have listed below the references that I suggest, in a logical reading sequence. If you belong to the Users Group, you may already have all but two of these documents: [06] [20] [27] [15] [25] [33] [34] [07] [09] [44] [29] [18] [41] [26] [21].

[01] author unknown, "Data Base Retrieval Optimization", SCRUGLETTER, Vol. III, No. 5, 1979.

[02] author unknown, "FASTIO", S.E. newsletter, date unknown.

[03] author unknown, "FASTIO Benchmark Benefits", SCRUGLETTER, Vol. III, No. 5, 1979.

[04] author unknown, "Slides on optimizing" [??], CCRUG Meeting minutes, distributed January 1981.

[05] Keith Baer, "ARHND, Virtual Array Handler", HPGSUG 1980 San Jose Swap Tape.

[06] John Beckett, "Managers, You Can Control Response Time", HPGSUG 1980 San Jose Proceedings.

[07] Rick Bergquist, "Optimizing IMAGE: an Introduction", HPGSUG Journal, Vol. III, No. 2, 1980 (reprint from San Jose meeting).

[08] David Brown, "Disc File Access Optimization Using Multiple-Record, Non-buffered Data Transfer, SCRUG80 Proceedings.

[09] William F. Burggrabe, Jr., "Disc I/O Comparision Chart", HPGSUG Journal, Vol. III, No. 2, 1980.

[10] Stephen M. Butler, "Faster with Fast KSAM", HPGSUG 1978 Proceedings.

[11] Mike Casteel, "Programming for Multi-terminal Applications", HPGSUG 1980 San Jose Proceedings.

[12] Jim Dowling, "Performance Management Techniques for the HP 3000 Series III", paper presented to HPGSUG 1980 San Jose Meeting, not published in proceedings.

[13] EASY Software Company, "Blocked IO Reference Manual", 1980.

[14] Rick Ehrhart, "Using Extra Data Segments - Safe and Efficient", HPGSUG 1978 Proceedings.

[15] Robert M. Green, "Principles for Optimizing Performance of On-line Programs", HPGSUG Vol. II, No. 2, 1978 (also printed in Denver meeting proceedings).

[16] Robert M. Green, "SPL/3000 in a Commercial Installation", training guide published by Robelle Consulting Ltd, 1980.

[17] Robert M. Green, "SPL/3000: Overview and Common Errors", HPGSUG Journal, Fall 1979.

[18] David Greer, "Checkstack and Controlling COBOL Stacks", HPGSUG 1980 San Jose Proceedings.

[19] David J. Greer, "Memory Files", unpublished paper of Robelle Consulting Ltd.

[20] Dick Hamilton, "Tips for News Users", HPGSUG 1980 San Jose Proceedings.

[21] Hewlett-Packard Co., "Application Design and Optimization for the HP 3000", internal training manual [see your S.E.].

[22] Hewlett-Packard Co., "COBOL/3000 vs. COBOL II Performance", 1980.

[23] Hewlett-Packard Co., "Performance and Optimization Seminar for NOWRUG", 1979.

[24] Marc Hoff, "Using Extra Data Segments", HPGSUG Journal, Vol. I, No. 4, 1977.

[25] Jack Howard, "Extra Data Segments and Process-handling with COBOL", HPGSUG Journal, Vol. II, No. 1, 1978 (reprint from SCRUG78).

[26] Jack Howard, "System Design and Optimization Techniques and Tools", HPGSUG Journal, Vol. III, No. 3, 1980 (reprint from SCRUG79).

[27] John E. Hulme, "System Performance and Optimization Techniques for the HP/3000", Applied Cybernetics Inc., 224 Camino del Cerro, Los Gatos, CA, 95030, 1980.

[28] Steve Kaminsky, "KSAM vs. IMAGE", HPGSUG Journal, Vol. I, No. 6, 1978 (reprint from SCRUG78).

[29] Madeline Lombaerde, "NOBUF/NO-WAIT I/O", HPGSUG 1980 San Jose Proceedings.

[30] Eugene H. Mitchell, "IMAGE Access Timing Test", HPGSUG Journal, Vol. III, No. 2, 1980.

[31] Christine Morris, "FORTRAN Optimization", HPGSUG Journal, Vol. I, No. 6, 1978.

[32] Gary B. Nordman, "Using a Hierarchical Data Structure", HPGSUG 1980 San Jose Proceedings.

[33] Alfredo Rego, "Design & Maintenance Criteria for IMAGE/3000", HPGSUG Journal, Vol. III, No. 4, 1980 (reprint from SCRUG80).

[34] Bernadette Reiter, "Performance Optimization for IMAGE", HPGSUG 1980 San Jose Proceedings.

[35] Robelle Consulting Ltd., "SUPRTOOL User Manual", 1981.

[36] Joe Schneider, conversation with the author regarding MPE IV/Series 44 experiences and other topics, January 1981.

[37] Joe Schneider, "TBPROC, Table-handling with Extra Data Segments", HPGSUG Contributed Library Volume 7.

[38] Anil Shenoy, "Sort Performance Guidelines", S.E. Note 170, October 16, 1979.

[39] Rodney Smith, "Application Design for HP 3000", SCRUG80 Proceedings.

[40] Ed Splinter, "Optimizing FORTRAN", HPGSUG 1977 Proceedings.

[41] Jim Squires and Ed Splinter, "System Performance Measurement and Optimization", HPGSUG 1978 Proceedings.

[42] Chuck Storla, "Determining File Usage", paper presented to HPGSUG 1980 San Jose Meeting, not published in proceedings.

[43] Mark Terribile, correspondence with the author, January 1981.

[44] Mike Vislosky, "FASTIO", HPGSUG 1980 San Jose Proceedings.

[45] Geoff Walker, "IMAGE Optimization Checklist", HPGSUG Journal, Vol. II, No. 2, 1978.

[46] Dave Walmsley, "RPG/3000 Program Optimization", HPGSUG 1977 Proceedings.

[47] Frederick White, "Improving Performance of IMAGE Applications", HPGSUG Journal, Vol. II, No. 4, 1979 (reprint from SCRUG79).

[48] Ted Workman and Mats Jonsson, "The Effect of Disc I/O on System Performance", unpublished paper presented to HPGSUG International Meeting in Switzerland, September 1980.

[49] Lu Yamada, "DBLOAD times with varying BUFFSPECS", S.E. newsletter, date unknown.