JULY/DECEMBER 1982
VOL. 5, NO. 3,4

JOURNAL

OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED

JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED
JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED
JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED
JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED
JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED
JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED
JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED

CONTENTS

JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Editor's Note

This issue of the Journal represents a departure from our usual practice of presenting a series of user-related articles. One major feature article is presented with two reviews by user group members. This approach provides an indepth look at one idea and appropriate comments from our membership.

We will continue to present relevant material for the membership's consideration. Your suggestions and/or comments are always appreciated.

John R. Ray

This paper represents my personal concepts of programming and performance as best I can express them at this time. Nothing I have either included or excluded is meant to represent the official posture of the Hewlett-Packard Company.

Hewlett-Packard allows, expects, and encourages individual initiative. The existence of this paper exemplifies that philosophy in action. Hewlett-Packard has completely supported my efforts. I have enjoyed total freedom in my choices of subject, content and format.

This paper originated from the Baltimore, Maryland sales and service district office of the Eastern Sales Region. It was physically prepared on equipment in the Technical Center in Rockville, Maryland.

The graphics were created on a 2647A terminal using Interactive Formating System software. Textual data, sample programs excepted, were prepared using HPSLATE. Final printing was done on the 2680 Laser Printer. A COBOL program using IFS intrinsics generated input to the printer.

I have had much help and cooperation in preparing this paper. Thanks to all with special appreciation to Ruth, my wife, for her patience, support and unselfish sacrifice of many evenings and weekends during its preparation.

Jim May

SECTION 1 GENERAL INTRODUCTION

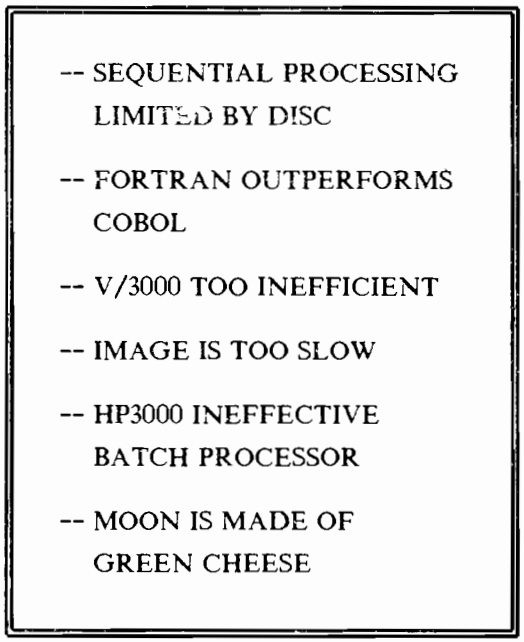

Figure 1

TRADITION (Figure 1)

-- SEQUENTIAL PROCESSING IS LIMITED BY DISC ACCESS

This is one of those generally accepted truths that all of us have been taught from the beginning of our careers. In most cases, on most machines, this is probably a valid generality. Because HP has concentrated on transaction processing, certain defaults built into the file system often make this assumption invalid when evaluating batch processing performance.

-- FORTRAN OUTPERFORMS COBOL.

In typical business applications, this is generally untrue. The FORTRAN compiler is admittedly superior in handling numeric data if we limit the definition of numeric data to binary and real data formats. The COBOL compiler, however, does a much better job with ASCII numeric data and is quite effective when dealing with files, records, and ASCII character fields. The net effect is that COBOL usually outperforms FORTRAN in the average commercial application.

-- V/3000 IS TOO INEFFICIENT.

Considering the powerful, highly generalized capabilities provided within V/3000, this statement is not acceptable. The original V/3000 did have design characteristics, particularly the use of KSAM and the prohibitive form file recompilation techniques, that left a bad taste in our mouths. The new V/3000 has corrected these shortcomings. In any particular application, a good programmer could probably outperform V/3000; even so, V/3000 is a highly efficient subsystem that deserves the chance to earn our confidence.

-- IMAGE IS TOO SLOW.

IMAGE is not at all slow but it is easily abused. IMAGE is an excellent example of a network data base and as such is inherently very rapid when used for record retrieval. Unfortunately, the price of rapid random retrieval is relatively slow structure maintenance, particularly when the structure becomes complex. Add to this the overhead for sophisticated internal security, multiuser update access, and extensive chain sequencing and you have a heavily burdened environment. In effect, the demands of our applications lead to slow performance; IMAGE itself is not inherently slow.

-- HP 3000 IS INEFFECTIVE IN BATCH PROCESSING.

The HP 3000 running MPE is admittedly biased towards transaction processing. The defaults built into MPE have not been chosen to maximize batch processing; the programmer who attacks a batch application without taking this into consideration may be displeased with the results. An informed programmer will know which override options to invoke in order to bring out the batch processing strengths of the machine. When handled properly, the HP 3000 is capable of surprising batch performance.

-- THE MOON IS MADE OF GREEN CHEESE.

I finally stopped believing this in 1969.

PROGRAMMING	FOR	PERFORMANCE
-- Programmer is problem solver		-- Measured by your MANAGEMENT !!
-- Programming is problem solving		-- Compromise between elements
-- Entire problem is fair game		-- Doing job -on schedule -within budget
-- Entire solution is fair game		-- Response (TP & B)
		-- Throughput (TP & B)

Figure 2

PROGRAMMING AND PERFORMANCE (Figure 2)

-- PROGRAMMING

A programmer is much more than a technican who encodes a problem solution in machine readable form. A true programmer is deeply involved in formulating the solution to the problem and, in some cases, may even identify and define the problem prior to compounding a solution.

No discussion of programming, therefore, can be limited strictly to an examination of computer coding techniques. The subject must be broadened to include all aspects of the problem and all details pertinent to the problem solution.

-- PERFORMANCE.

We technicians often forget that the criteria for performance measurements are defined by management and may differ greatly between organizations and even between functional areas within an organization. In too many cases we refuse to accept the fact that the only satisfactory solution may require compromise, most often a sacrifice of technical elegance, in order to meet a management objective. A technically advanced solution finished too late may be worthless; one that exceeds the planned cost may be even worse.

Fortunately for us technicians, this discourse will concentrate on technical performance. We will be concerned with traditional indicators, response and throughput, in both batch and transaction processing environments. Many times, one can be gained only at the expense of the other. Luckily, some techniques can improve performance in all instances.

PERFORMANCE CURVE (Figure 3)

A generally acceptable graphic depiction of a machine's performance is a curved line, the "performance curve", showing the gradual performance degradation for the average program as the machine is progressively loaded. In typical transaction processing environments, the curve shows a definite pattern (Curve #1). At first, performance degrades very little as the first few interactive jobs compete with one another. As more jobs are added, each tends to have a

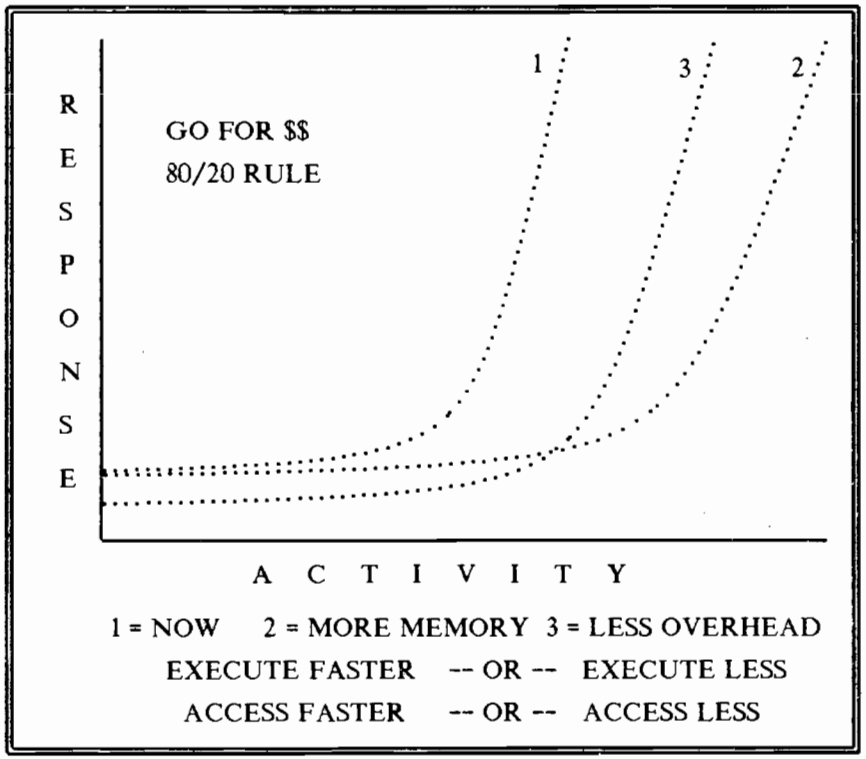

Figure 3

greater negative impact and the curve begins to climb more rapidly. Eventually the "knee in the curve" is reached where each added job causes a disproportionate degradation. At the knee the machine has usually reached the point where the aggregate useful work done by the machine drops for each job introduced.

If the real memory in the machine increases, performance normally improves as shown by Curve #2. On the low load end, the performance improves only slightly if at all since low load performance is not generally memory limited. Mid-range performance improvement is more noticeable and the mid-range itself is extended. The knee still occurs but does not show up until the machine is more heavily loaded. The usual impact of additional memory shows up more in improved throughput rather than in improved individual program

response.

Increasing the raw execution power of the machine affects performance differently from increasing its memory. Improvement shows up immediately on the low end of the curve and the knee shows up later. Similar improvement can also be attained by improving mass storage access even without increased raw execution power. Curve #3 shows a typical example of such improvement.

Improved programming exhibits characteristics similar to those of Curve #3. We could expect this since improved programming usually causes less code to be executed or less disc accesses to be made. This, in fact, becomes a basic guideline for improving programming. Most effective performance improving techniques center around reducing executed code and reducing disc access. Since improvements in disc access are more practical to accomplish and will reduce code execution as a byproduct, disc access reduction usually assumes first priority.

TRANSACTION PROCESSING PERFORMANCE (Figure 4)

Batch performance is fairly easily measured. We can easily time how long a job runs, what resources it seems to be absorbing, and how much competing batch jobs inhibit one another. TP performance is much more difficult to quantify.

TP performance is measured by two yardsticks, throughput and response. Throughput is the objective count of the number of transactions that can be processed in any given time period. Response is more difficult to measure because it is a subjective evaluation.

A programmer has more influence on response than on throughput. Additionally, dramatic changes to response may make little change to throughput. This paper will concentrate on response.

TP is too complex to try to examine as a single entity. For simplicity, I am limiting this overview to an evaluation of

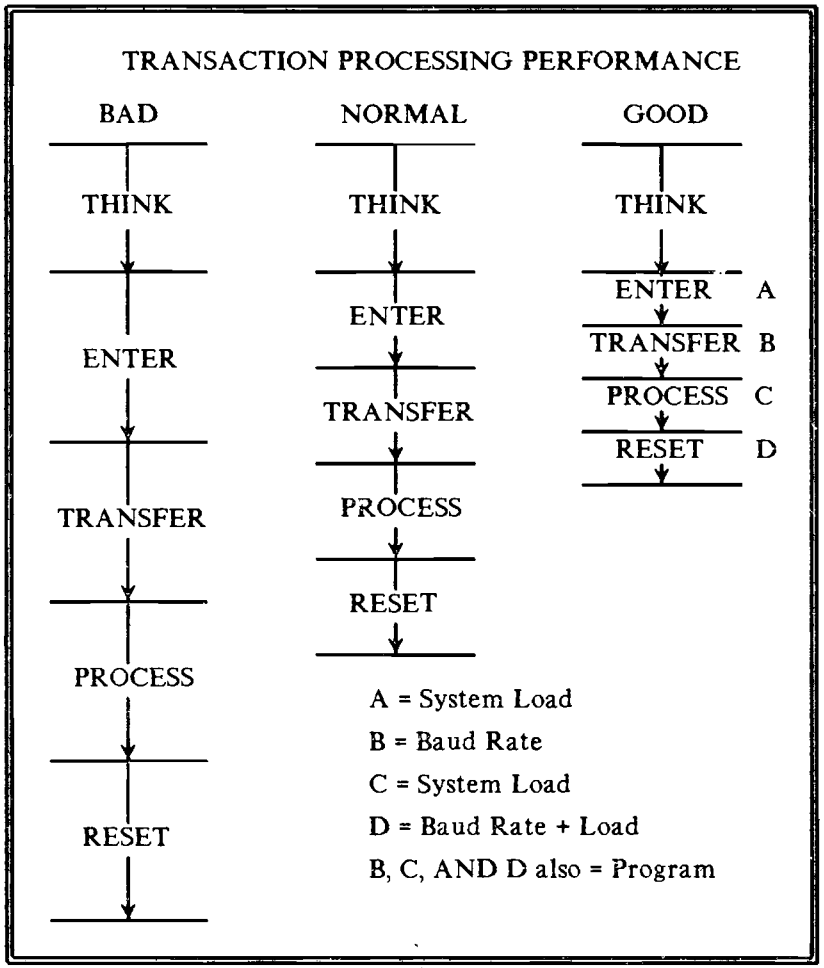

Figure 4

the processing of a simple transaction. This can be broken into 5 distinct pieces of time.

1. Think Time.

The time for the user to fill a screen. Think time is normally the longest item in TP and is application dependent. I will arbitrarily disregard think time in this paper.

2. Enter Time.

The time from the hitting of the enter key to the beginning of actual data transfer to the machine. This time is mostly determined by the hardware, the operating system, and the overall machine load. Although a programmer may impact enter time, we will not discuss it.

3. Transfer Time.

The time from the start of data transfer from the screen to its completion. This is mostly a hardware function. In some instances the programmer can change this item. We will address transfer time briefly.

4. Process Time

The time your program spends actually processing the screen input. By definition this is under programmer control. We will concentrate on improving process time.

5. Reset Time

The time required to prepare the terminal for the next user interaction. In pure data entry this time might be trivial. In other applications it might involve screen switches, response transmission or similar lengthy activities. We will examine some cases involving reset time.

Traditional measurements of system response times tend to emphasize the measurement of enter time. For system evaluation this is a valid point of reference. We programmers must concern ourselves with application dependent factors over which we have some control.

Programmers deal most directly with transfer, process, and reset times. These three items are the primary criteria used by users in measuring how responsive a system is.

The user who waits fifteen seconds for the computer to digest a screen of data and prepare the terminal for next input feels that the machine is not responsive. Our job is to make

the machine responsive in the eyes of the user.

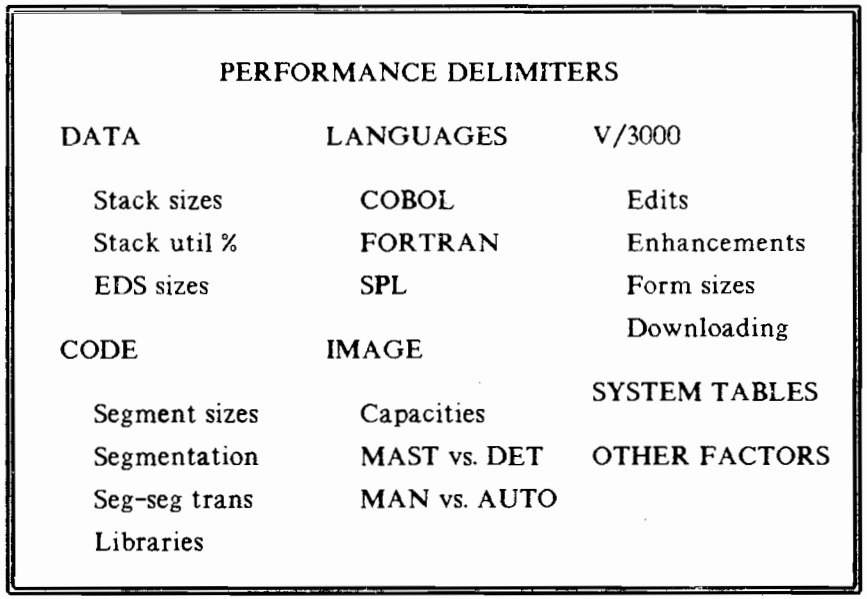

Figure 5

PERFORMANCE DELIMETERS (Figure 5)

Almost anything can impact performance. I have consciously oversimplified the situation by dividing the subject into seven groupings. Each of these groupings could be considered most important by any individual. This would be influenced considerably by a person's experience and background. A quick overview of each will set the tone for the presentation of my views on the subject.

GROUPS 1 AND 2 -- DATA AND CODE.

These related items are very important factors influencing performance. They have also received much attention by many people.

This is fortunate in that the programmer has definite

guidelines to follow to try to avoid creating totally unacceptable programs. It is unfortunate in that we often assume that the volume of verbiage on a subject indicates its relative impact on our work. I feel this is not always true and that some of us have become entirely too concerned with data and code considerations.

I am almost certain that neither data nor code are the primary culprits when the first questions I am asked about a program are "Do I have my stacks small enough?" or "Have I segmented the program properly?". All too often these are dead giveaways that the programmer has become intimidated by the massive documentation about code and data and has failed to get a good perspective on the whole situation. How, for example, could code and data be causing 10 second response delays in a program that makes 250 data base accesses per response?

I firmly believe that code and data can significantly impact performance. I believe even more firmly that they should be held suspect only after many of the other possible contributors have been reviewed and evaluated.

GROUP 3 -- LANGUAGES.

This is another area where I feel we have all expended too much effort. Except for special cases in which specific capabilities of a language are required, I have yet to find a program which would be meaningfully more efficient in one language than another. The one rule I would accept would be the prohibition of interpretive BASIC in a production environment.

I have seen many instances where the absolute requirement to have a "COBOL shop" has denied a programmer access to efficiencies available in other languages, particularly SPL. I have seen even more cases where the fear of the assumed inefficiencies of COBOL has resulted in FORTRAN or SPL programs which quite often run slower than a COBOL version.

Although this may represent a minority viewpoint, I claim

that COBOL will often be the most efficient and the most effective language in the typical commercial application. And even if this may not be true, I further claim that the choice of languages does not have a meaningful impact either way. If you like COBOL, use it. If you hate COBOL, use FORTRAN or perhaps SPL. In any case, use whatever helps you as an individual to get your job done.

GROUP 4 -- IMAGE

By now, everyone should be wondering just what I think are the important performance delimiters. We've finally hit one. IMAGE has a big effect on performance.

The factors sublisted in this group are the ones usually quoted as being critical. These and the use or non-use of sorted chains have been discussed to death. In addition, with the probable exception of the constraints on sorted chains, their importance is generally blown all out of proportion.

We will cover the impact of IMAGE but it will be done from a different vantage point. Our primary concern will be centered around the ways we have structured our data bases and the effects those structures have on performance.

GROUP 5 -- V/3000.

V/3000 is very good but it is not perfect. As with any highly generalized package, V/3000 can probably be "beaten" in any given application by a highly skilled programmer. In some applications, V/3000 may also be a less desirable choice because more performance oriented but less generalized techniques are available. A case in point might be the data capture environment where we would want to consider using the more efficient data capture intrinsics.

Just because V/3000 may not always be the best solution in all instances should not become a rationale for avoiding V/3000 altogether. V/3000 simply has too many capabilities to be ignored. It is also more efficient than most of us

probably realize.

V/3000 was introduced with two specific faults which gave some of us some bad memories. Both of these have been corrected. The KSAM oriented forms file structure has been replaced by a vastly superior file access method. The ability to recompile only modified forms has greatly reduced development and maintenance overhead. If we have not reviewed our evaluation of V/3000 since its introduction we may be cheating ourselves.

There are some capabilities still suspect within V/3000. I claim that in most cases, the culprit is the heavy demands we build into our applications rather than the way V/3000 handles those demands. I strongly suspect that most of us would be pleasantly surprised at the performance V/3000 gives compared to that provided by user written code performing the same functions. We would also probably be appalled by the volume of code we would have to write and maintain to replace standard V/3000 capabilities.

There is much potential benefit for us if we closely evaluate our techniques of using V/3000. In many cases we can improve performance by using V/3000 differently. In the average application, however, we would probably do much harm by trying to avoid or replace V/3000.

GROUP 6 -- SYSTEM TABLES.

Let's handle this fast. Look at system tables from an overall system point of view. Forget about them in applications programming.

GROUP 7 -- OTHER FACTORS

When somebody ends a list of items with a group called "other factors", they probably plan to quickly dismiss those same factors as relatively unimportant. In our case, I have deliberately placed them last for emphasis. What many might consider relatively unimportant are the very items experience has taught me to look at more carefully. I think that look

will be most revealing.

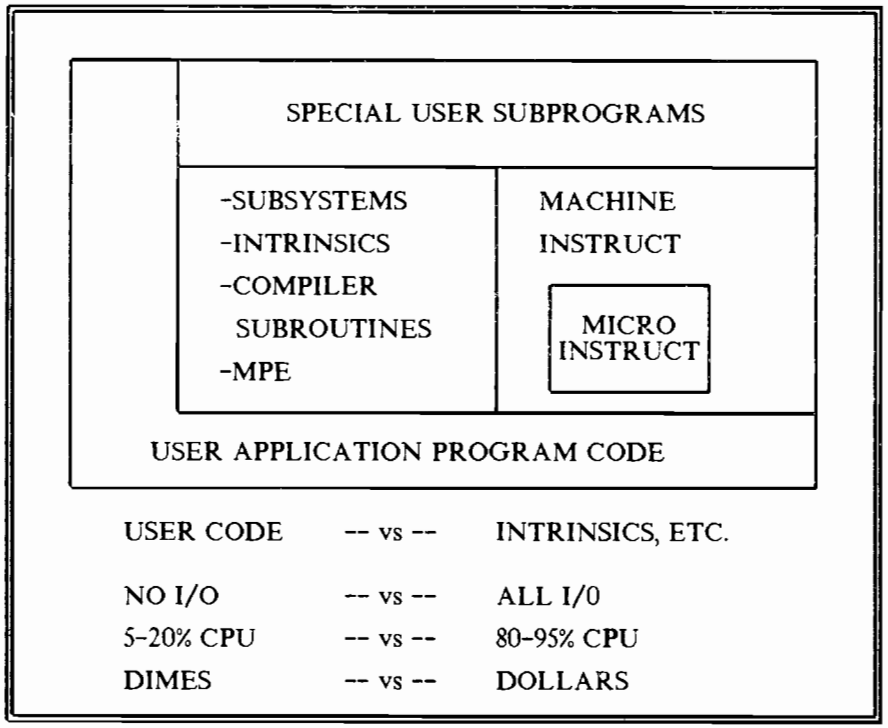

Figure 6

PROGRAM CHARACTERISTICS (Figure 6)

How we visualize our programs can have a great influence on our attempts to improve their performance. A reasonably accurate picture of a typical commercial application program might surprise some of us. It might also help explain why two programs, one written by a highly technical programmer and the other written by an experienced but relatively non-technical programmer, can have maddeningly (to the technical programmer) similar performance characteristics.

The most important fact to be realized is that the normal application program written on a typical modern commercial

computer is a "driver". Whenever we write a program, particularly in a high-level language, we do not generate computer instructions. We are actually writing compiler instructions which will be converted into computer instructions. Our choices of coding techniques can have significant influence on some of the code generated. In general, however, the problem being coded has a far greater effect than our choices of how we code our solution.

A consequence of the "driver" aspect of our programs is that our program code (that is, the portion controlled by our coding techniques) never performs I-O. Except for rare, highly specialized, privileged mode applications, all I-O is performed by MPE Intrinsics. Once we have designed our application characteristics, the I-O required is essentially independent of our programming language or our programming techniques.

Some programmers could be somewhat discouraged at being told their coding techniques have relatively little effect on a program. Others will be relieved on hearing the same message because it allows them to code without fearing that they might mess up a program through "poor" programming. Both of these types of programmers have missed the boat on performance.

Better programmers reprioritize their efforts away from mere technical coding competence and concentrate on design. They realize that the characteristics designed into an application are the major determinants of performance.

Being better programmers, we will concentrate on design in our search for improved performance. Even relatively small changes in design can have more effect than massive changes in pure coding. Wise decisions during application design can have immense impact on eventual performance.

SUBPROGRAMS (Figure 7)

Utilizing subprograms allows us to program for performance. Period. End of sentence.

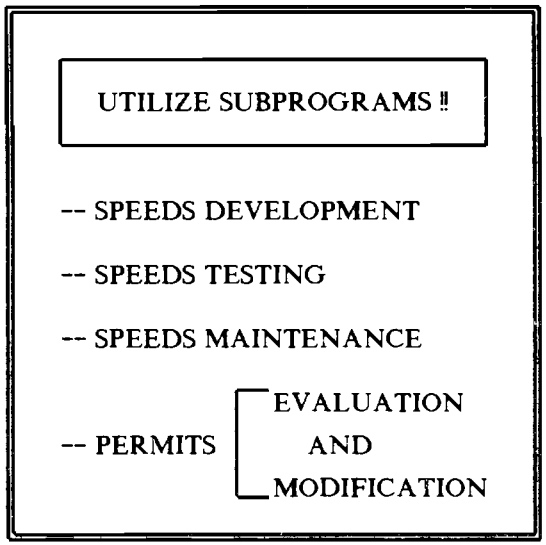

Figure 7

A non-trivial program written using subprograms will be faster to develop and test than an equivalent program written as a single unit. This is not prejudiced conjecture by me; it is a widely recognized fact. Since development and test time for a program are included in the broad definition of performance, programs written in subprogram form give improved performance.

Maintenance is also simplified for programs built from subprograms. It is easier to determine where to modify a subprogram than where to modify a unit program. The validity of the change is more easily tested in a subprogram. Subprograms even require less expense to test than do unit programs. Smaller listings, shorter compilations, and more controllable logic give us tremendous return for our investment.

Subprograms would justify themselves solely on the merits claimed up to this point, but we should look deeper. Subprograms can be powerful tools in the attempt to improve application performance.

Subprograms are prime examples of modular program and application architecture. Modularity isolates functions so that modifications affecting those functions can likewise be isolated. Isolation of modifications allows more accurate evaluation of the effects of those modifications. The more accurately we evaluate our modifications, the more effective our modifications can become.

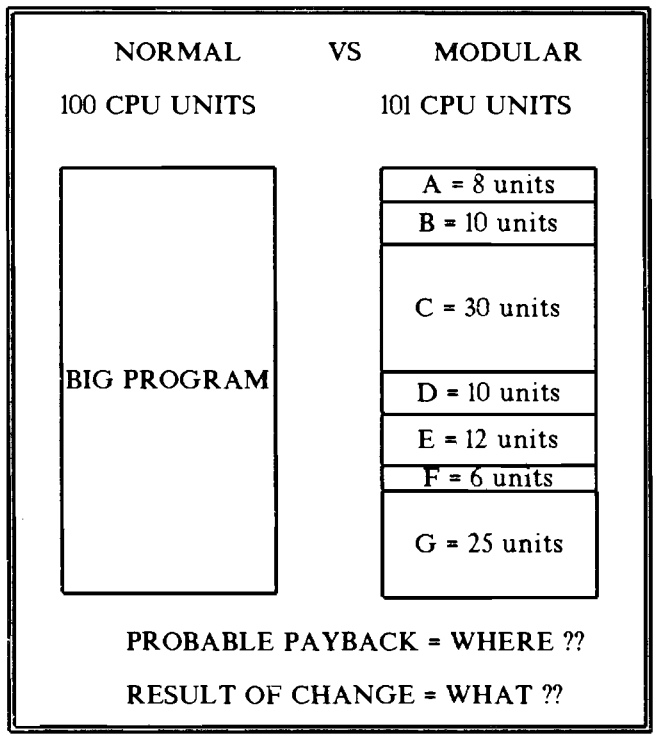

Figure 8

MORMAL vs MODULAR (Figure 8)

Subprograms do not give us more efficient programs. All factors being equal, a unit program will execute more efficiently than one written in subprogram form. Subprograms, however, give us the means to improve performance and efficiency in ways generally unavailable in

unit programs.

Improving performance comes only partly from improving programming. Far more important than how much we improve programming is where we decide to try to improve programming. I would much prefer to reduce a high overhead item than a low one.

Intelligent attempts to improve performance require a disciplined plan of action. That plan must include at least four discreet items:

-- Evaluation of existing performance
-- Identification of candidates for alteration
-- Selection and implementation of changes
-- Evaluation of resultant performance

To get a feel for the potential importance of subprograms for performance improvement, we can review two hypothetical cases. The first involves a unit program; the second, a modular program. In both cases, someone with clout has decided that the performance needs improvement.

CASE I -- THE UNIT PROGRAM.

The programmer follows a rational plan:

-- Measures performance of unmodified program. 100 CPU units are needed for control run.
-- Uses past experience to identify probable bottlenecks
-- Program changes, all quite valid, to improve performance at selected areas
-- Measures performance of modified program. 75 CPU units are needed for test run.

Now we have a few questions to answer:

Q. Was 25 CPU unit improvement good, fair, or poor?
A. Can't tell.

Q. How much of the potential improvement was realized?

A. Can't tell.

Q. Did every change improve performance? A. We don't know. We couldn't make evaluations of each change because the compile cost was too high.

Q. What do we try next? A. Whatever the boss says.

This would have been so easy had the original changes taken us from 100 CPU units to 20. That type of improvement gets praised, not questioned. But who said everything was easy?

CASE II -- THE MODULAR PROGRAM.

This programmer also follows a rational plan:

-- Measures performance of unmodified program. 101 CPU units are needed for control run. Usage per module ranges from 6 to 30 units.
-- Decides that most probably payback is in modules C and G with 30 and 25 unit loads, respectively
-- Uses past experience to decide that module G is most likely candidate.
-- Program changes to module G
-- Measures performance of modified program. 86 CPU units are needed for test run. Module G has gone from 25 units to 10 units.

This programmer also must answer some questions:

Q. Was a 15 CPU unit improvement good, fair, or poor?
A. Quite good. The portion changed showed a 60% (from 25 units to 10) reduction.

Q. How much of the potential improvement was realized?
A. For module G, probably most of it. But we only changed 25% of the program. 75% still merits evaluation.

Q. Did every change improve performance? A. Looks probable.

Q. What do we try next? A. Module C is a likely candidate because it absorbs 30 units during execution. That's more than a third of the remaining overhead!

We may require a number of iterations before we reach the point of diminished returns. At least we have a better way to tell where we have probably reached it. In addition, we have begun to build a body of experience to help us optimize our next program more easily.

SECTION 2 BIG ITEM CHECKLISTS

Talking about techniques to use in isolating performance problem areas is valuable. Knowing what to do next is equally important. This paper will attempt to identify some of the most frequent "next steps".

Sometimes, of course, the "next step" becomes the only step. This occurs most often when we are called upon to help optimize a unit program whose performance is suspect. We can't waste time wishing the program could be more easily analyzed. After all, if it were easily analyzed, we wouldn't have been called in. So we take it as we find it.

Every experienced performance consultant has a mental checklist of potential problem areas. This checklist includes specific techniques found helpful in the past and the expected benefits for each.

Each checklist is different. The differences depend upon the consultants background, track record, and personal biases. A specialist experienced in commercial applications has a different checklist from a specialist who has worked with technical applications. Similarily, checklists based on batch applications will differ from those written for on-line systems.

My background is in commercial applications, both batch and on-line. I would like to share part of my checklist with you. The sequence is for convenience and continuity; it has no priority implications. For each major item, I will organize its analysis this way:

-- A header visual showing:

- TASK: The description of the checklist item.

- PLAN: The proposed corrective action most likely to succeed.

- GOAL: The expected benefits.

-- One or more subordinate visuals showing.

- Why the item might be degrading performance.

- Detail examination of the proposed action.

- Why the proposed action should improve performance.

TASK: REVIEW SEQUENTIAL FILE PROCESSING

PLAN: REBLOCK FILES
USE NOBUF I-O

GOAL: REDUCE DISC ACTIVITY
REDUCE CPU CYCLES USED

Figure 9

SEQUENTIAL FILE PROCESSING (Figure 9)

Almost every shop has batch programs that process large sequential files. Quite often they are programs originally

written for Brand-X and converted to run on the 3000. They frequently run much slower than we think they should.

We keep reminding ourselves that most commercial applications are I-O bound, not CPU intensive. We begin wondering about the power of the 3000 when our "I-O Bound" programs run at close to 100% CPU utilization. It's time we found out why this happens.

TASK: Check out sequential files looking for high record volumes, low blocking factors, and default file access.

PLAN: Increase blocking factors and replace default file access with nobuf I-O.

GOAL: Reduce disc activity (blocking factors). Reduce CPU load (nobuf I-O).

I-O -- BUFFERED vs. NOBUF (Figure 10)

Those of us who learned programming on a Brand-X machine, know how sequential files are processed. A pair of buffers are set up in the program and the physical I-O system tries to keep them full. The logical I-O system provides records to us by indexing through the buffers as I-O is requested.

This indexing through internal buffers is extremely efficient. Machines using this approach to sequential processing usually handle batch processing better than on-line processing.

Every operating system on every computer is optimized for a particular environment. This includes both the internal architecture of the I-O system and the choice of standard defaults for its user interface. HP emphasizes on-line processing and has designed its I-O system accordingly. Batch processing is performed well but suffers somewhat to benefit on-line work.

On-line processing emphasizes random retrieval. Random retrieval implies retrieval of a record from a reasonably

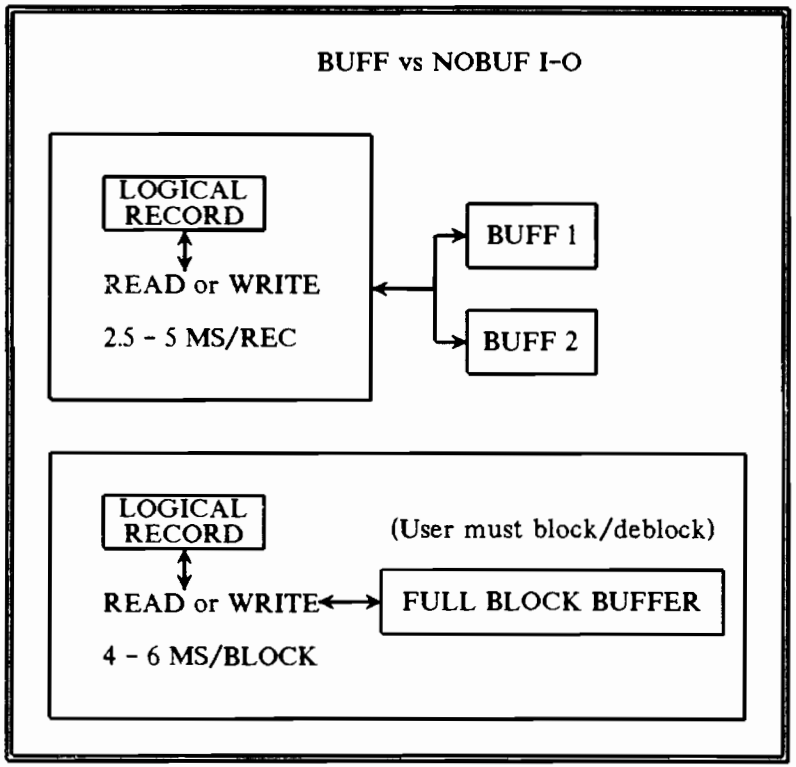

Figure 10

well-defined location on disc. Knowing a records location eliminates most of the benefits of large blocking factors. Therefore, HP has set up relatively small default blocking factors. These utilize disc space well but are usually comparatively small for batch processing.

On-line processing also requires effective file sharing capabilities. Files are quite difficult to share when the I-O system puts the buffer inside programs. HP simplifies file sharing by isolating the buffer from the program. This is excellent for on-line activities but increases overhead when doing batch.

File buffers reside in extra data segments under MPE. Logical I-O requires the file system to expend considerable

effort to transfer records back and forth between the user stack and these extra data segments. This explains the high CPU load during sequential file access.

Once we know how the defaults in MPE increase overhead in batch applications we can make intelligent adjustments. The rewards are well worth the effort. In typical cases, we can reduce overhead by 70 to 90 percent.

Blocking factors are easily changed by coding the "REC" parameter in the "BUILD" command. I can't tell anyone what factors to use but I would probably choose between 7 and 30 depending on record size.

Conversion to nobuf I-O is not so simple but contributes most to CPU load reductions. There are two basic ways to do this:

-- Code your routine directly into your program. Although this is how I coded my sample program, I prefer the second technique.

-- Code your routine in a subprogram. I prefer this technique. It suits my mode of operation..

RECORD SORTING (Figure 11)

Every shop needs sort capabilities. Batch applications are particularly heavy users of sorts because they are inherently sequence dependent.

Sorting places heavy demands on the machine. Although we cannot reasonably eliminate sorting from our programs, we have techniques to reduce their overhead.

TASK: Review our use of the sort capability in our environment.

PLAN: Avoid file-to-file sorts and the use of the stand-alone sort subsystem.

TASK:	REVIEW USE OF SORT SUBSYSTEM
PLAN:	AVOID SORT SUBSYSTEM AVOID FILE-FILE SORTS
GOAL:	REDUCE DISC ACTIVITY REDUCE CPU CYCLES USED

Figure 11

GOAL: Reduce I-O activity in sort functions. As a secondary benefit, reduce CPU load on the system.

BRUTE FORCE (Figure 12)

When you had a small machine and tape was your primary storage media you learned how to drag data through programs using brute force. The standard mode for sorting was to read a tape into the sort and write sorted records back to tape. Sometimes this had to be done in multiple passes with multiple tapes. It wasn't much but it certainly beat loading and unloading card hoppers.

Modern computer systems support and use tape but they rely more often on disc for primary storage. Modern programmers have also begun to rely on disc. They have finally gotten rid of their little drawers full of punched cards. Why, then, do they still do their work, particularly their sorts, by brute force.

The stand-alone sort is useful and necessary. It is also a resource hog. The I-O required for a sort is extensive, especially when added to the I-O to read and write output

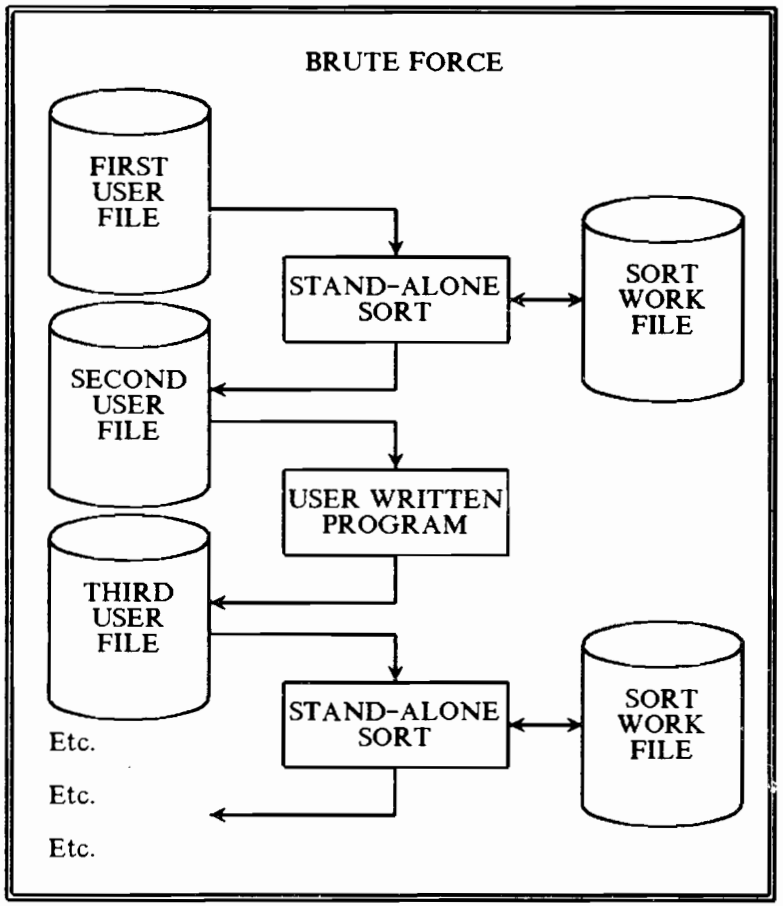

Figure 12

files.

Disc availability is a prime constraint on performance. Sorts, particularly the stand-alone file-to-file sort, eat deeply into this availability. In the interest of performance we should try to reduce these activities whenever practical.

FANCY BRUTE FORCE (Figure 13)

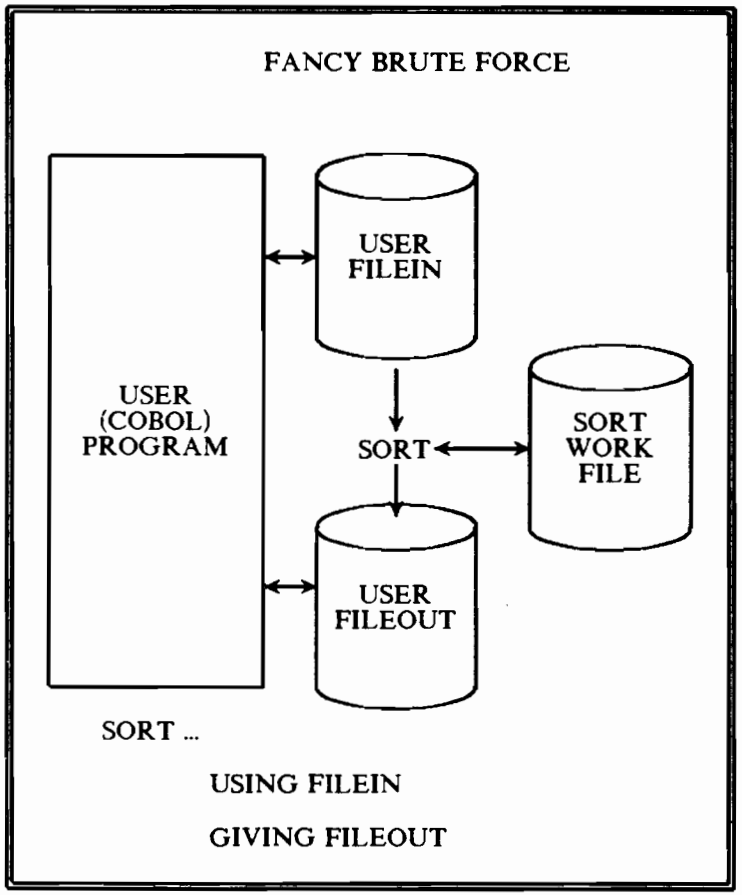

Figure 13

Programmers have learned how to invoke the sort programmatically. They have effectively used this to replace two or more programs and one or more sorts with single programs. Then they have held back from using the true capabilities of programmatic sort access.

Programmatic file-to-file sorts are not necessarily bad but they usually have a negative effect on performance. They use up disc resources at at least the same rate as stand-alone sorts. Except for very small data volumes, programs written with file-to-file internal sorts usually run slower than the

individual programs and sorts they replaced.

Some shops will not allow programmers to use sort capabilities programmatically. I think they are missing a good opportunity by this blanket condemnation. On the other hand, if they are only avoiding "fancy brute force", they can be partially excused.

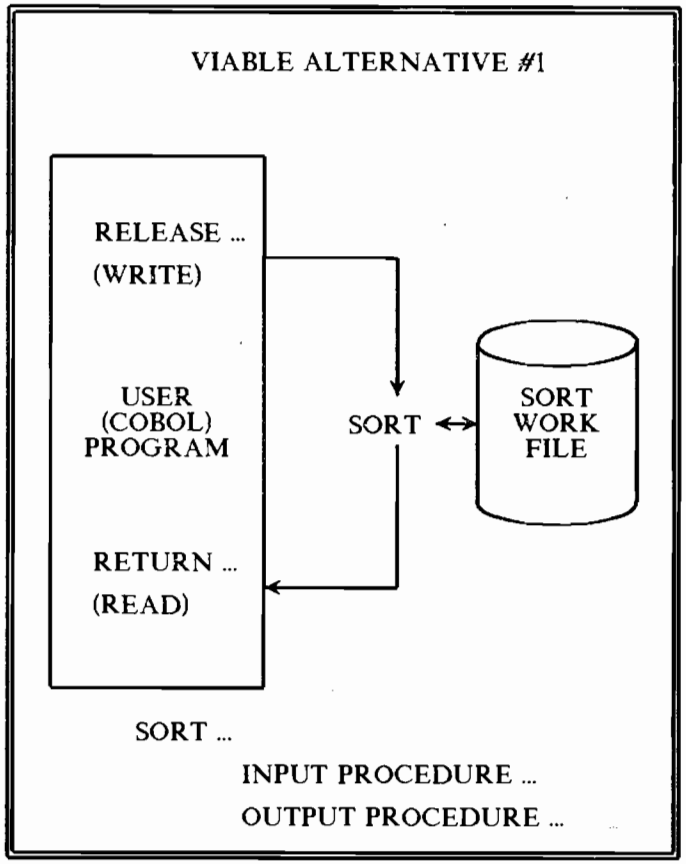

Figure 14

VIABLE ALTERNATIVE #1, DIRECT SORT INTERFACE (Figure 14)

Programmatic access to the sort subsystem gives the

programmer many attractive options. Hooks built into the sort allow programmers to pass records directly to the sort and receive sorted records directly from the sort. This capability can be used effectively to improve program performance.

Direct interaction with the sort allows us to avoid disc activity. Every time we interact directly with sort we avoid two potential disc accesses; we have eliminated a read access and a write access. This can greatly reduce disc activity.

We also reduce CPU overhead by talking directly to the sort. We get rid of the CPU overhead for the file system to process our logical I-O. Unfortunately, there is a price to be paid for this ability.

Interaction with the sort requires direct resources other than disc I-O and CPU cycles. The sort needs memory to be efficient and that memory comes from the user stack. If our stack is too small or the program data is large the internal sort may lose its value. It may then become a burden.

Another limitation of the internal sort is the inability to have multiple sorts executing simultaneously. In programs with multiple internal sorts we may have to allow some file-to-file sorts or the logical equivalent.

VIABLE ALTERNATIVE # 2, PROCESS HANDLING (Figure 15)

MPE offers us another means to reduce sort overhead. We can use process handling as a means to bypass I-O in sorting applications.

Some shops fear process handling. I wish more of them could begin using it to advantage. Perhaps there is too much emphasis on the "special" in "special capabilities". Whatever the reason, many of us look upon process handling as a tool only for exceptional cases. We should view it as an exceptional tool useful in many applications.

There is no justification for reserving process handling

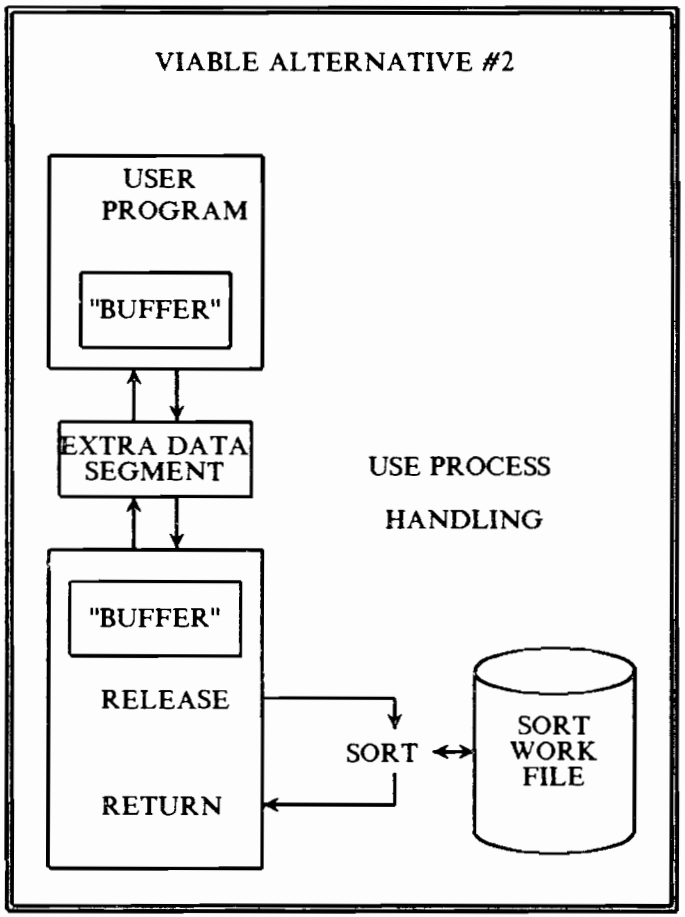

Figure 15

solely for esoteric or multi-threading environments. Process handling is perfectly suitable for use in relatively simple, single-threaded applications. An obvious use would be in a program involving sorts where we can use multiple processes to advantage.

We should examine how sort functions can be accomplished using process handling. The concept is similar to the normal programmatic sort interface especially if we isolate all calls to the process handling intrinsics in a subprogram. The rest is simplicity itself.

1. The master program initiates sorting by a call to the subprogram. The subprogram creates and activates a slave program whose job is to sort records. The subprogram also creates an extra data segment to use in passing blocks of records to the slave. It waits until the slave program is ready for work.

2. The slave begins life by starting up its own internal sort. Its subprogram then wakens the master, in effect saying "OK, I'm ready". It waits for instructions.

3. The master sends raw records to the slave for sorting. Every call to the master subprogram represents logical passage of a record to the slave. The subprogram fills an internal buffer with records.

4. When the subprogram has a full buffer it loads it into the extra data segment and wakes the slave, effectively saying "OK, give these to sort". It waits until the slave has done its work.

5. The slave retrieves the extra data segment, unloads the logical records, and passes them to the sort. When finished, it wakes the master, saying "OK, I'm ready for more". It waits for more.

6. Steps 3, 4 and 5 are looped through until all records have been sent to the slave. The master then sends an "end of data" message to the slave after preparing to receive sorted records. The master waits now.

7. The slave receives the "end of data" message and lets actual sorting begin. When sorted records are available, the slave is ready to awaken its master.

8. The return of sorted records from master to slave is a reversal of the process described for sending unsorted records from master to slave.

A sample program is included with this paper to show the technique. It does not use subprogram interfaces because I wanted to isolate the example in one source file. In real

life, I would recommend a subprogram.

This example obviously reduces disc activity. It offers other benefits as well.

1. The slave has its own stack which is not loaded with application data. The sort can be given plenty of room to breathe. The slave can perform actual sorting more efficiently than could its master.

2. The program is no longer limited to having only one sort active at any time. Except for the normal constraints of MPE, any number of sort slaves may be active at any time.

As with other alternative techniques, process handling extracts a cost.

1. Process handling absorbs CPU overhead. If records are not passed back and forth in blocks, the overhead may be relatively high.

2. Multiple processes and their stacks need memory. This may cause problems in some cases. Your machine size and workload profile are critical decision criteria.

3. Process handling is not difficult but is more complex than the use of standard compiler features. Subprograms eliminate this problem once you have the subprograms written and tested.

DATA VALIDATION AND CONVERSION (Figure 16)

Both batch and on-line programs perform extensive data validations and conversions. Quite often these account for a high portion of the overhead within a program.

Validations and conversions come in two basic flavors- algorithmic and tabulated.

-- Algorithmic validations check validity according to a processing rule. Check-digit calculations and pattern

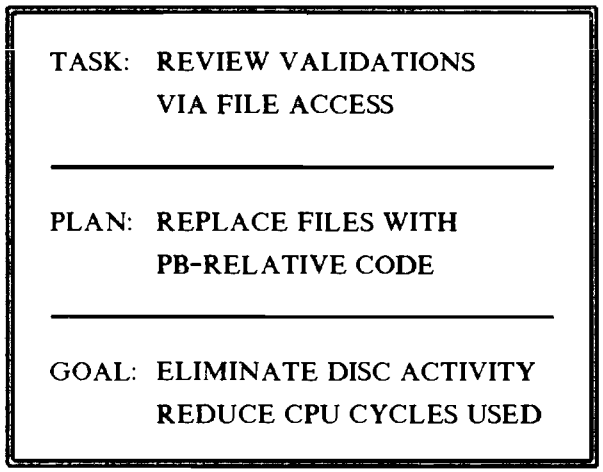

Figure 16

matches are examples.

-- Algorithmic conversions convert data based upon a conversion algorithm. Julian to Gregorian data conversion is an example.

-- Tabulated validations check validity by searching a table or file for a "hit". Customer validation through attempted retrieval against a data base is an example.

-- Tabulated conversions convert data from argument to result by searching a table or a file. Converting numeric error codes to meaningful error messages is an example.

Algorithmic techniques and table oriented searches are normally efficient but may be difficult to modify and maintain. File oriented techniques are easily modified but absorb considerable overhead. Performance considerations may make file oriented techniques too expensive.

TASK: Review file oriented validations and conversions.

PLAN: Replace files with PB-relative code. This normally takes the form of a binary search procedure.

GOAL: Eliminate disc I-O completely when possible. Significantly reduce CPU overhead.

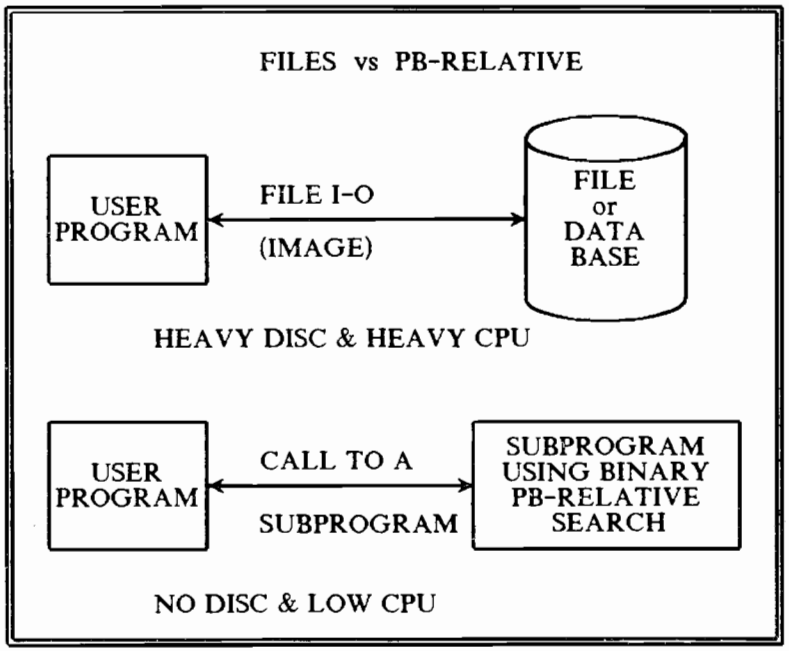

Figure 17

FILES vs PB-RELATIVE (Figure 17)

Files are frequently used for data validation or code expansion in all data processing. We usually use KSAM or IMAGE. In either case the disc I-O is extensive and involves considerable overhead.

In many cases there is no viable alternative to file access. Volatile information is ideally suited to randomly accessed and maintained file structures. Massive quantities of data cannot be economically maintained except on disc media.

In other cases we may be able to use techniques that require little or no disc access. The following are the normal techniques used.

1. Tables may be hard-coded into storage if the information is not too massive and is not volatile. Massive data volumes are prohibitive and volative data values create maintenance hang-ups.

2. Volatile tables may be filled at execution time via file access. This is impractical for large files and may cause excessive overhead in short duration programs.

3. Values may be hard-coded into program code as literals. This saves stack but creates maintenance problems for volatile values.

4. Data values may be loaded into extra data segments. This is especially useful for highly volatile data values in transaction processing applications utilizing multi-threaded process handling. It is a relatively complex approach but can be of great use.

Another available but seldom used technique is to place data values into PB-relative code. The code is SPL and the internal retrieval is via a binary search. This technique has tremendous potential for improving performance.

1. Disc access is eliminated.

2. CPU overhead for retrieval is extremely low.

3. Storage demands are relatively low, particularly if the code resides in readily shareable SL segments.

4. Access is simple since a single call statement is sufficient.

Nothing is free. For all its benefits, PB-relative code has many drawbacks.

1. Dynamic changes cannot be made to PB-relative code.

2. SPL is a requirement for PB-relative binary access.

3. CST table limitations may restrict use of easily shareable SL files.

On balance, I strongly recommend PB-relative techniques in cases where performance potential is needed. I also recommend that they not be applied just because they are available. Like many other techniques, its use must be based upon its relative value within the application.

Two example subprograms are included in this paper. One demonstrates retrieval of fixed length values; the other, variable length. Obviously, either will validate data arguments.

PB-RELATIVE MAINTENANCE (Figure 18)

A major reason to avoid using PB-relative code for validation and conversion purposes is the difficulty it presents to maintenance. The potential benefits cannot, however, be ignored.

Maintenance of tabular data in PB-relative code is not trivial but it need not be excessively difficult. The first step requires that records be prepared for batch input to a maintenance program. This is a common function in all batch maintenance applications and should be no problem.

The actual maintenance run is different depending on how we wish to apply the maintenance. There are three primary approaches to applying maintenance.

1. Maintenance data can be converted to CON (constant) constructs in SPL. These can be inserted into a skeleton program which will be compiled into a USL. This is a straight-forward approach but requires multiple steps.

2. Maintenance data can be applied directly to a generalized USL skeleton. This is quite efficient but requires considerable knowledge of USL structure.

3. Maintenance data can be applied directly to either an RL

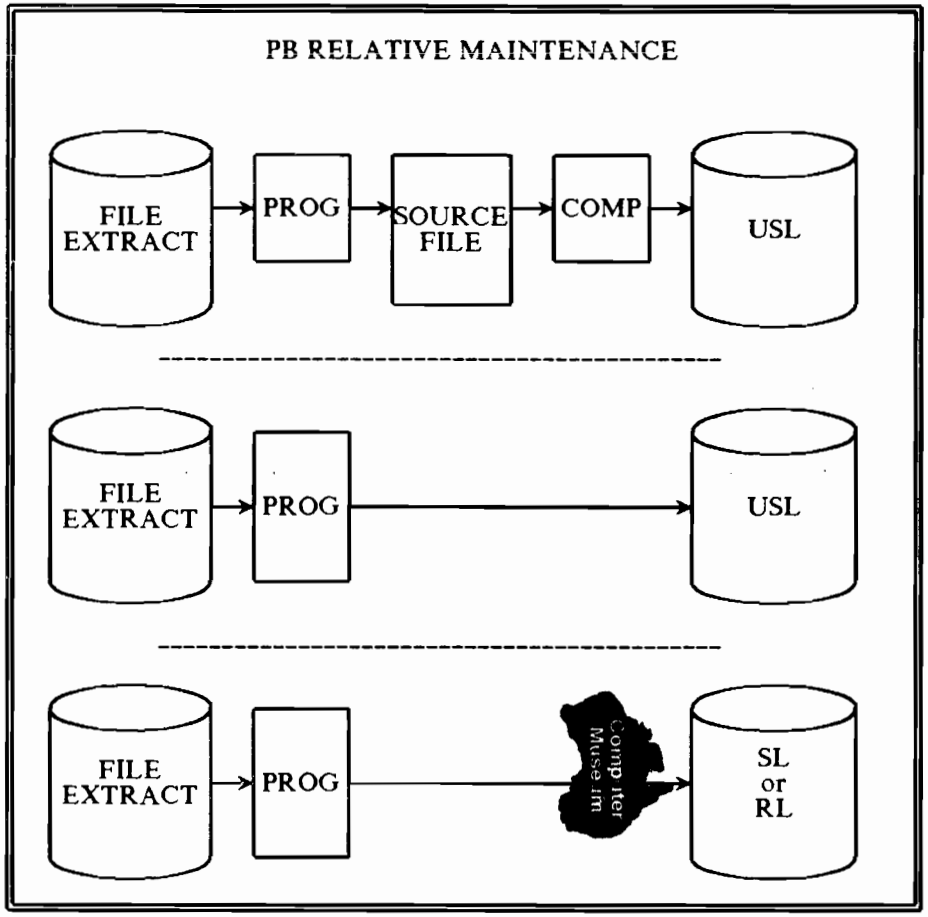

Figure 18

or an SL. This is even more direct than modifying a USL but is probably more difficult.

We have a fourth technique that I hesitated to put on the diagram. We can change the program directly. I left it off the diagram because it destroys my credibility with the fragile types.

Program code is not user-alterable during execution. That does not keep us from altering files just because they happen

to represent programs. There are many things you can do with program files if the need arises. Keep your eyes open for opportunities. It's fun.

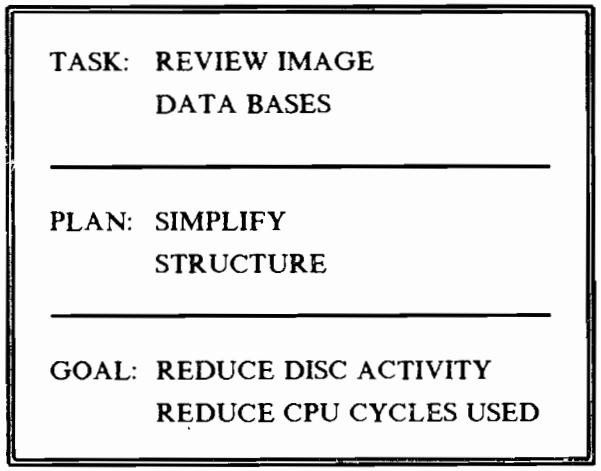

Figure 19

IMAGE DATA BASES (SIMPLIFICATION) (Figure 19)

Commercial applications contain requirements that often lead to complex data structures. IMAGE gives us the means to control these data structures with reasonable ease. Sometimes we forget that what is reasonable for us from a design and access point of view may be totally unreasonable at the machine performance level.

Designing paths to detail data set records is a simple task, particularly if the path is from an automatic master. The desire for rapid random retrieval often justifies this structure.

Using a path from a detail data set to a manual master can provide non-programmatic data validation as well as potentially rapid retrieval. This capability justifies many of our decisions during application and data base design.

When performance becomes unsatisfactory we have to revalue our decisions. With experience, most of us become more selective with the facilities we include in our designs. We have learned that complex, aesthetically pleasing structures may become our white elephants of performance.

TASK: Review IMAGE data base structures.

PLAN: Revalue our design with hopes of simplifying the structure without undue impact on functional performance.

GOAL: Reduce disc activity and CPU load.

REDUCE LINKAGE PATHS (Figure 20)

IMAGE gives excellent performance in random record retrieval. This is one of the major reasons why IMAGE has become so widely accepted. We appreciate the capability to read detail records by multiple key items. Sometimes we get carried away in our appreciation and go too far.

Rapid detail access is achieved using pointers which allow precise record location. These pointers have to be created before the record is accessible. The price of rapid retrieval is paid by the machine when it sets these pointers.

The majority of the overhead for adding a detail record linked to multiple masters comes from establishing the linkages to those masters. On a dedicated Series III you can closely predict that about 7 or 8 linkages can be created or deleted per wall second. This has tremendous implications for performance.

Adding or deleting a detail linked to 7 masters will take about 1 wall second. This is the main reason why IMAGE reloads and batch record maintenance programs run relatively slowly. With 7 linkage paths per detail you can only expect about 4000 adds and deletes per hour.

Transaction processing is also severely impacted by the

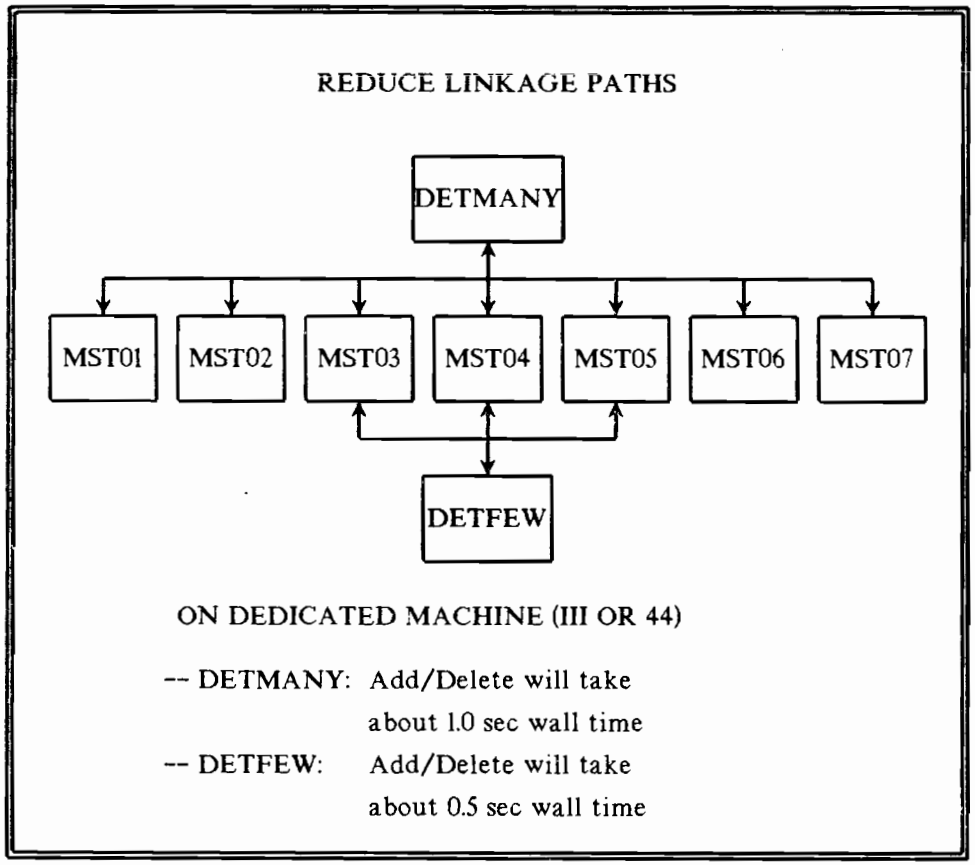

Figure 20

maintenance of linkage paths. If a transaction causes additions of 3 records and each has 7 linkage paths you have every reason to expect at least a 3 second delay during those adds. Under normal circumstances when others are sharing the machine you should not be surprised with much longer delays. This helps explains the following common situation. You can supply the interpretation.

- Design goals call for 5 second maximum processing time delay

- Testing shows excellent average delays of 3.5 seconds

- Initial production delays average a satisfactory 4.5 seconds

- 3 months later the delays are averaging 10 seconds

- You have a problem. The test cases did not predict the realistic performance.

Worst case performance comes during record modification if a search item needs to be changed. This requires a physical delete and add. The time for this change is the sum of the delete and add times. This type of processing can break the back of an application.

All too often the ability to define multiple paths into details seduces us into defining too many paths. Unless the path is required or gives a high priority extra capability you should think seriously before creating it.

1. Every path defined has essentially the same cost.

2. Every path probably does not give the same payback.

3. Is the payback worth the cost?

4. Will the extra path create the monster called "change = delete + add"?

Limiting the number of linkage will improve performance. Consider the case where we go from 7 to 3 linkage paths.

1. Processing delays will drop from 1 to about .5 seconds.

2. Delete and add for change will occur less often and will take only half as long.

3. Delays on a loaded system will probably grow much less and will be less dramatic.

4. Reloads and batch maintenance will be significantly faster.

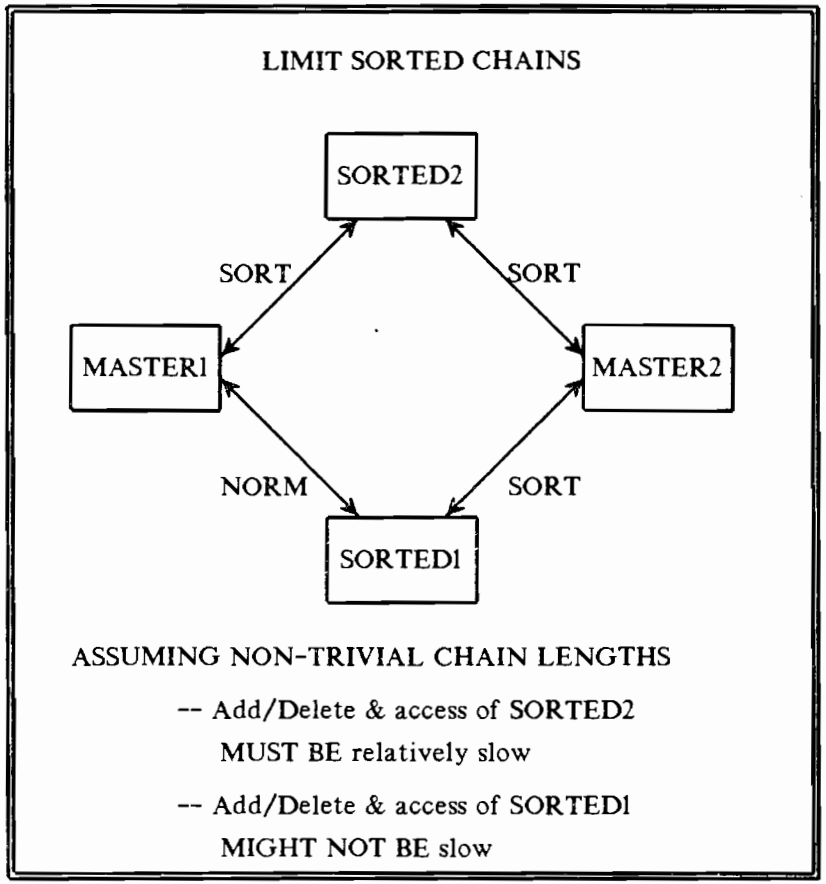

Figure 21

LIMIT SORTED CHAINS (Figure 21)

IMAGE allows details within a logical path to be sorted by some other item value. This is a powerful capability which may be of use to us. It is also an expensive capability.

Addition of a detail with a sorted path causes more overhead because sort sequence must be maintained. IMAGE starts at the logical high end of the chain and follows it backwards until it finds the correct logical home for the detail. It then links the detail into the logical path and goes about its business for the remaining linkage paths.

If the detail fits at the logical end of the chain, the added overhead is trivial. If sort-item values are random, IMAGE must read half through the chain on average to find a logical home. This can be quite expensive.

Note: Sorted paths can give you unpleasant surprises.

1. The value sorted extends from the sort item to the end of the record. This is the "implied sort".

2. Additions will use the entire implied sort value.

3. Any sort before addition should include the full implied sort item. Hideous performance can result if this is not done. This explains why sort items should usually be at the end of a record.

4. Implied sort sequences can be useful. They are also functionally dangerous. IMAGE allows update in place on non-sort items. The implied sequence disappears once updates are done to fields within the implied sort.

Defining multiple sorted paths within a detail guarantees performance degradation. There is no way to avoid extra overhead with multiple sorted paths.

LIMIT LONG OR VOLATILE CHAINS (Figure 22)

The ability to chain logically related records together is a necessary function in any Data Base Management System. This function is designed with relatively short logical chains in mind. It is not meant to be abused.

A good logical chain usually maintains relationships based upon important data items. They are designed to preserve solid logical relationships among records.

1. Customer identities linked to their orders.

2. Line items within an order linked to an order header.

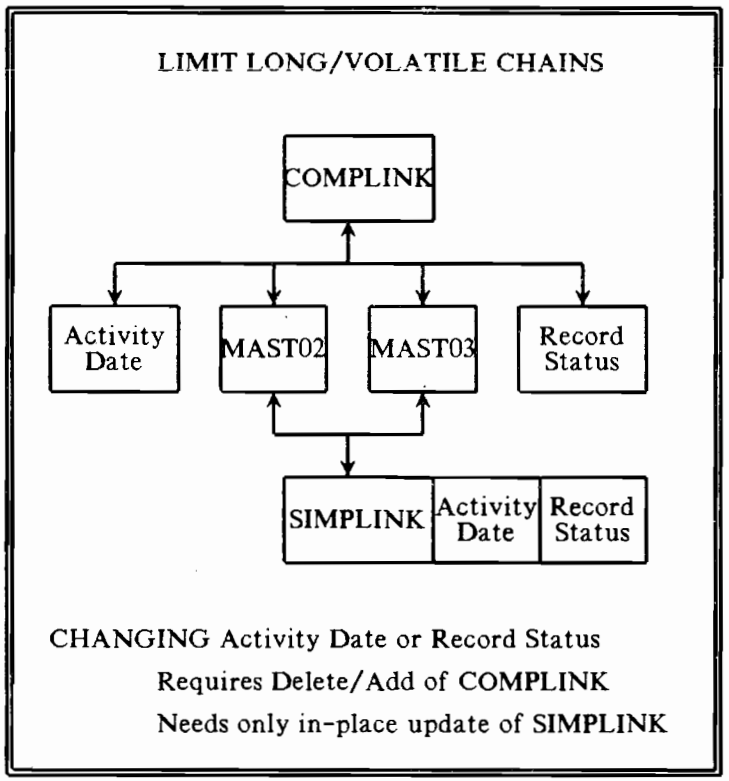

Figure 22

3. Order line items linked to inventory status records.

4. Inventory status records linked to open purchase orders.

5. Purchase orders linked to responsible vendors.

A poor logical chain usually attempts to maintain relationships which are relatively less important. They are often designed to try to create artificial order out of inherent chaos. Even worse, they may exist simply because the capability to create them is provided.

1. Personnel records linked to employee sex.

2. Invoice records linked to invoice status.

3. Student records linked to grade values.

4. Inventory records linked to last activity dates.

Another form of poor chain would be the case where the search value is volatile. Every time the value changes, a complete delete/add must be done. This can be very expensive.

My generalized definition of a poor logical chain is based upon present technology. A poor chain is one whose cost exceeds its value. When technology reduces cost sufficiently, I will change my definition.

The definition of a poor chain is also not absolute. If your application benefits more from a chain than it spends to have the chain, the chain is good. Good and bad are merely a comparison of relative gain to relative cost.

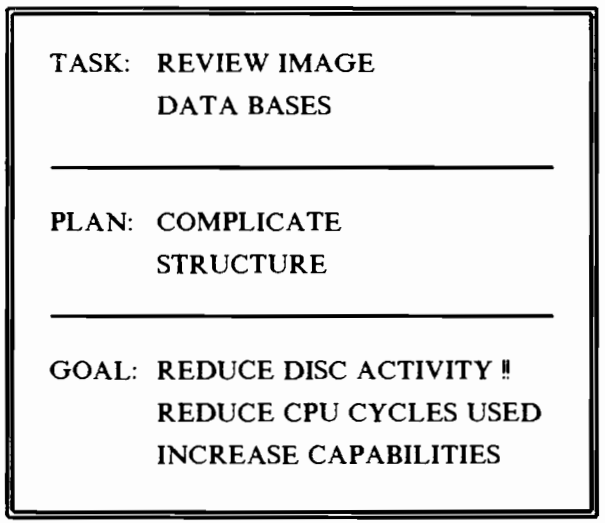

Figure 23

IMAGE DATA BASES (COMPLICATION) (Figure 23)

Simple IMAGE data base structures generally absorb less

overhead per function than do complex structures. This leads us to avoid complex structures to maintain good performance. Sometimes, however, the simple structure becomes a burden and causes unacceptable performance for some required application functions.

Overall performance within an application is a complex entity. Each function performed carries an inherent overhead based upon the function, how often it is performed, and the structure of the data base it accesses. Simple structures can degrade performance if they fail to permit efficient processing of frequently required functions.

Relatively complex structures may benefit overall performance if the structure matches the intended use. Simple structures, while inherently more efficient, may degrade performance if they do not satisfy the application. We cannot judge nor can we design a data base without extensive knowledge of the application.

We may find that our application cannot be serviced satisfactorily without complicating our data base. With proper planning we may be able to use complexity to our advantage to improve both functional and overall performance.

TASK: Review IMAGE data base structures.

PLAN: Revalue our design based upon our knowledge of the application and try to find places where a more complex structure can improve performance.

GOAL: Selectively increase complexity to make our application overhead go down when performing required functions.

SELECTIVE CONSTRUCTIVE ABUSE (Figure 24)

Simple data structures are normally preferable to complex structures. Chains should usually be avoided where data volatility is a problem. But you can throw out any generalized rule if it serves you poorly.

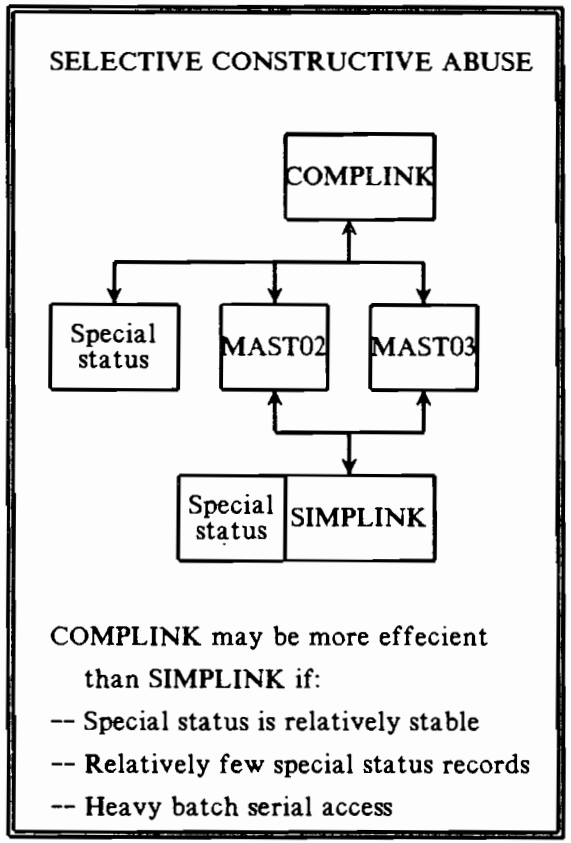

Figure 24

Every application is different. Its success depends much more on how well you have made it perform in the eyes of your users than on how well you have applied standard techniques and principles.

Sometimes you do everything "right" and create a performance dog. You may be stuck in the a lose-lose position.

1. Your application has two obviously good linkage paths. Performance is acceptable for all transaction processing.

2. Nightly batch exception reporting is horrendous. You can't get your special status reports finished by 8:00 A.M.

3. You establish a linkage allowing direct access by status. Note: You're lucky you didn't blow chain length limits.

4. Your nightly exception reporting is now a piece of cake.

5. You have a new problem. Transaction processing is dying. The extra chain and its volatility are eating resources alive.

6. Now what?

The time has come to revalue your position. Your thought and action process might go like this

1. I can only get batch efficiency using the complexity of an added linkage path.

2. I can only get transaction processing efficiency with a simple structure.

3. Maybe if I combine the two structures I can have the best of both worlds. I'll try a more complex physical structure that I can look at from two logical directions.

4. For my normal records I'll have a detail linked by the normal two linkages. As long as the status isn't really special I'll treat it as just another field.

5. For my special records, those with very special status codes, I'll design another detail linked to status.

6. With rare exceptions, my transaction processing will run well just like it did before.

7. My batch exception reports will be printed on time.

8. I think I've got it!

You have abused the data base by making it much more complex. You have been selective in matching your changes to your application. You have been constructive - your application now performs properly.

There are times to follow accepted rules and there are times to write your own rules. The only real problem is knowing when to do which.

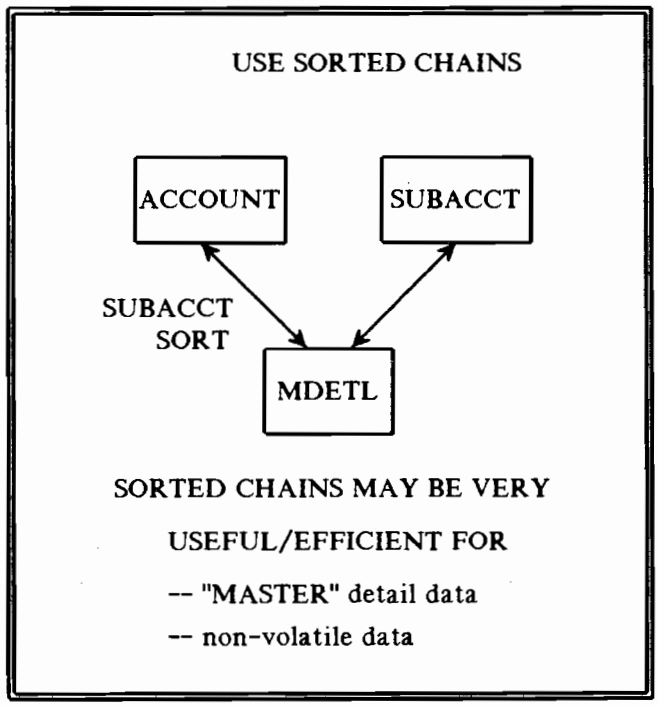

Figure 25

USE SORTED CHAINS (Figure 25)

Sorted chains are powerful tools and carry with them high potential overhead burdens. We should use them carefully but we should not be afraid of them.

Many data base designs could improve functionally if we

designed them with sorted chains. We avoid most potential uses because they will perform poorly. Certain cases will perform extremely well.

1. The detail sets are pure linkage sets: they merely serve to define ordered paths between masters.

2. The detail sets are "master" type details. They represent, for example, what might be considered a master record in an indexed file environment.

3. The data in question is stable.

This structure is extremely useful for keeping master implosion and explosion chains. In an accounting application, they can allow us to explode an account number into sorted sub-account to account.

CONSIDER COMPLEX SORT STRUCTURES (Figure 26)

Sorted chains and complex structures are both potential performance bottlenecks. Functional necessity may require their use.

IMAGE is not designed to give logically sequential access of master data sets. Indexed file structures provide logical sequential access but lack most of the capabilities of IMAGE. We're in that potential lose-lose situation again if we need IMAGE capabilities and have to have logical sequential access.

Some programmers solve this problem by maintaining dual data structures. They create the normal IMAGE data base and keep a separate KSAM file to contain key values extracted from their IMAGE master sets. It works well for most of them.

Other programmers attack the problem by setting out to emulate index type structures within IMAGE. I like this approach because it seems more fun to work with and it keeps everything in IMAGE. Once you define the master you follow this basic plan.

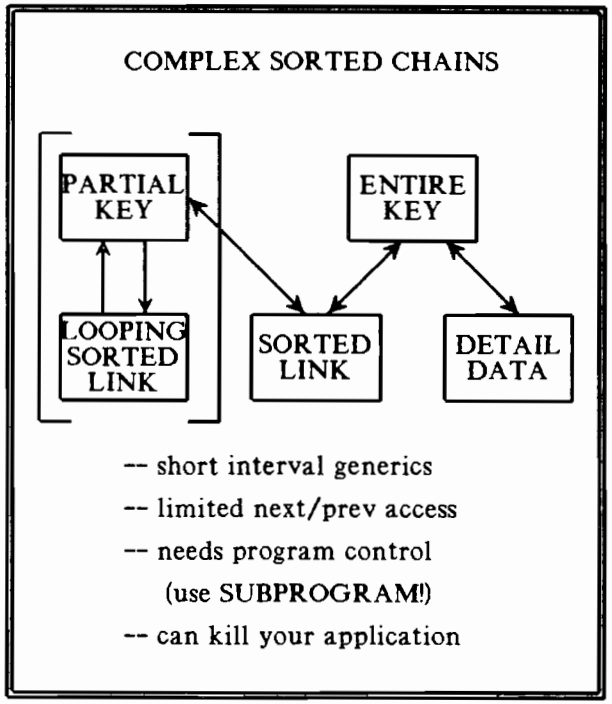

Figure 26

1. Define a path to a detail which links your master to your index emulation sets. This detail is optionally sorted by master value on the linkage path coming from the index emulators.

2. Define a detail index set to hold linkages between partial keys. This set links to your interface linkage detail and has two linkages to a detail linkage set.

3. Define a detail index set to hold linkages between partial keys in the master index set. One path is sorted to point to lower level (longer) partial keys. A second optional sorted path allows reversal in sequence.

4. A partial key of null values starts your emulated index.

5. The rest of the index structure contains gradually longer

partial keys. The common increment is two characters.

Index emulation has both good and bad points. You make your own choices since you pay the bills.

1. Short interval generics become possible.

2. Limited next/previous record capabilities are available. If you try to process the data base purely sequentially you will probably regret your decision.

3. This technique requires extensive programmatic control. I suggest a canned subprogram.

4. You may be tempted too much by logical sequential access. Remember that physical access will actually be totally random and expensive.

5. Your application may die.

TASK: REVIEW TP FUNCTION TRANSFER DELAYS
PLAN: ELIMINATE TRANSFERS AVOID BUILD/WRECK
GOAL: REDUCE OVERHEAD REDUCE DELAYS

Figure 27

FUNCTION TO FUNCTION TRANSFER DELAYS (Figure 27)

Batch processing applications tend to isolate functionally

similar records into groups (often as physical "batches") and pass them as a unit through a program or program stream. Pure transaction processing applications, on the other hand, provide users the ability to enter virtually any transaction with assurance that processing resources will be available. In theory, no transaction can be predicted before the user presents it to the application.

No matter how well we process a transaction, we are judged harshly if we take too long getting ready to process it. A beautifully performing program gives poor performance if it takes too long to begin executing. Our users will justifiably demand rapid transition between functions.

Transfers from one function to another can also involve considerable overhead. If this overhead is too high the machine can become so bogged down it has little left for its true job, processing transactions. Successful applications spend their resources processing, not getting ready to process.

We usually test programs for their ability to process transactions efficiently. We often forget that much more than efficient programs is needed for effective applications. We have to be concerned with function to function transfers if we expect success.

TASK: Review function-to-function transfers.

PLAN: Try to eliminate transfers if possible. When transfers cannot be eliminated, try to reduce their cost by reducing their overhead.

GOAL: Reduce system overhead in general. Reduce transfer delays to improve user perception of performance.

MAGNIFICENT MASOCHISM (Figure 28)

Transaction processing applications are often complex. Demands for extreme flexibility stretch programmers skills to the limit. Process handling is an easily implemented

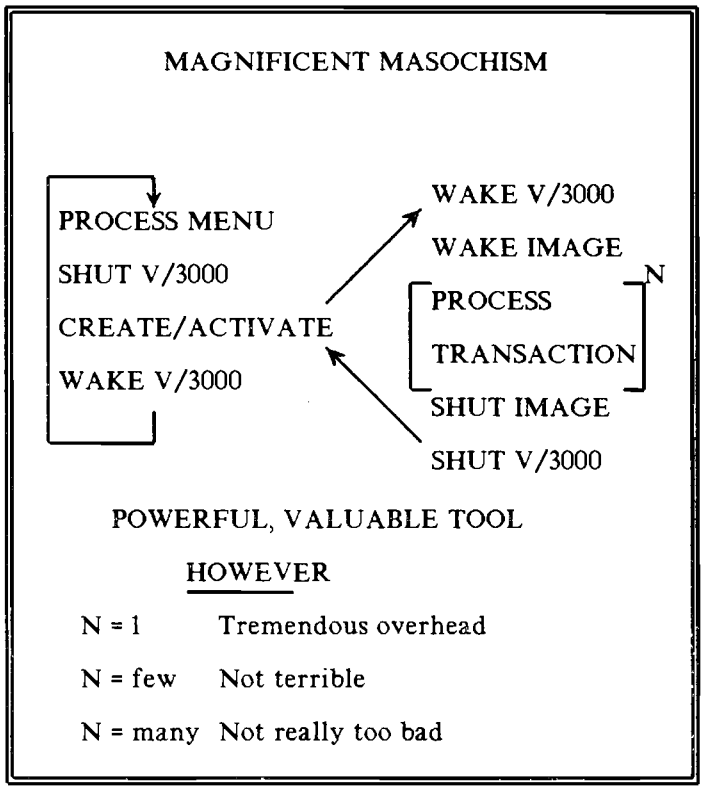

Figure 28

technique often used to help simplify a complex situation. It can be quite effective in the proper circumstances and can beat the machine senseless when misused.

Most users have plenty to keep them busy. They justifiably require the simplest possible interface between themselves and the machine. Most commercial software packages are menu driven to help win acceptance in the market.

Menu driven applications are easy to design and document. They can also be reasonably simple to program and test. Their biggest potential problem is a tendency toward tedious maintenance and relatively poor performance.

Modular programming techniques reduce the difficulty of

maintenance. The most modular technique involves isolating individual or similar functions in separate programs which can be controlled using process handling. A selection of a menu item triggers a programmatic execution of the appropriate program. In effect, the menu program issues predefined "run" commands which are functionally invisible to the user.

Function invisibility is not true invisibility. The user sees the execution of the internal "run" as a delay in processing. There is considerable overhead associated with both starting and stopping a program. The overhead to execute a normal transaction is usually much less than the overhead spent invoking its program.

When the selected program will execute numerous transactions before returning to the menu the overhead burden may be quite acceptable. As the number of transactions per execution gets smaller, the overhead becomes objectionable. If the design calls for only 1 transaction per execution the application is potentially terminally ill.

A standard technique to allow flexibility without tremendous overhead is to keep a program alive after it has been invoked. Process handling allows reactivation of a suspended program (process) with minimum overhead and delay. V/3000 processing may create some problems, however, and the limit on the number of processes alive within MPE may curtail your ability to use this technique.

VIABLE ALTERNATIVE (Figure 29)

Some programs promise so much power and flexibility that it is impossible to avoid some performance loss. With intelligent programming we can minimize the loss.

Subprograms allow almost as much flexibility as process handling and usually require less overhead. Critical values such as IMAGE and V/3000 control areas can be passed as parameters from caller to callee. A standard technique involves defining a single data area containing the most

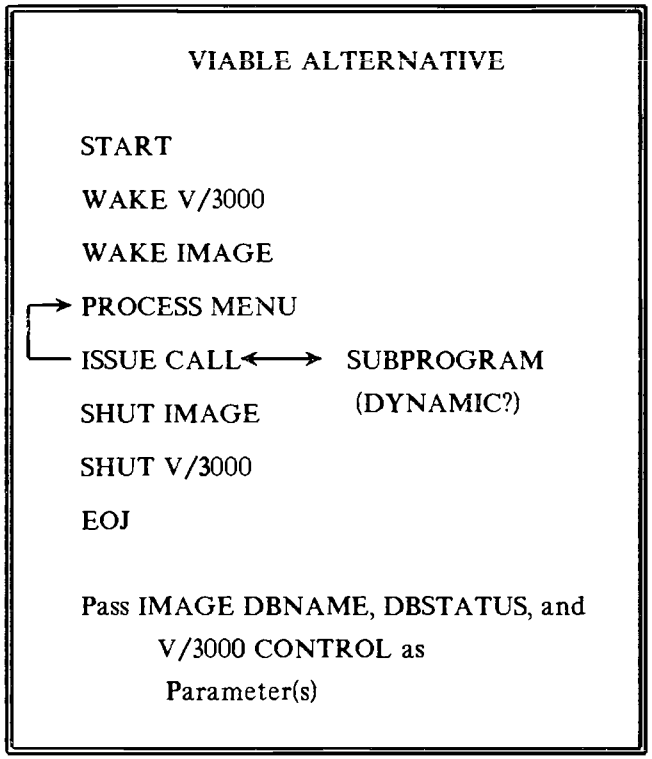

Figure 29

common data and passing it to all subprograms as a single parameter.

Stack sizes may be a limit on the use of subprograms. Dynamic subprograms can help control stack size but must be used carefully to avoid excessive overhead of their own. Excessively large stacks inhibit performance and so can excessive stack expansion, contraction, and initialization.

Dynamic subprograms also limit capabilities of the subprogram. Values kept in local storage is lost when you exit the subprogram. If files are opened they must be closed before you exit. More care is needed when you decide to make an existing subprogram dynamic than when you originally design it to be dynamic.

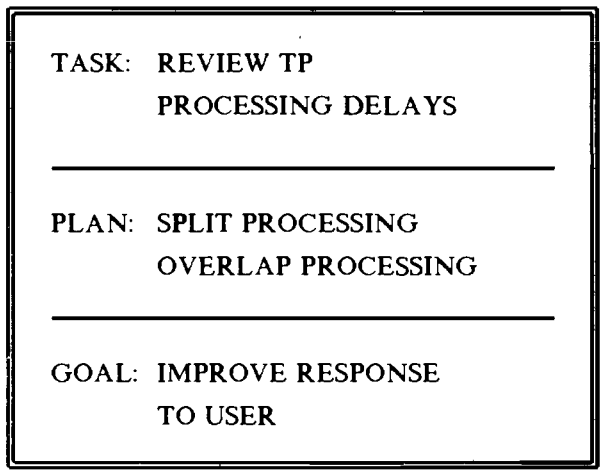

Figure 30

PROCESSING DELAYS (Figure 30)

Processing the actual transaction data is the eventual goal of transaction processing. The time taken for processing ranges from a very small percentage of the total transaction time to a very large percentage. When the percentage is too large we have to try to reduce it to a more acceptable level.

Processing normally breaks down into two phases, data validation and data storage or retrieval. Either or both phases may require extensive disc access with corresponding overhead.

Assuming overhead has already been minimized, there is virtually no way to do anything with the data validation phase. If we expect to reduce processing delays, we can make improvements only in the storage or retrieval phase.

MPE contains easily implemented techniques that allow us to improve performance as perceived by the user. Improvement in perceived performance will not reduce machine overhead and may in fact increase it. In the final analysis, though, increased machine overhead is meaningless so long as the user, the authoritative judge of performance, sees

improvement.

TASK: Review processing delays.

PLAN: Evaluate data storage and retrieval delays. When excessive, attempt to use MPE capabilities to reduce or eliminate them.

GOAL: Improve perceived performance as measured by the user.

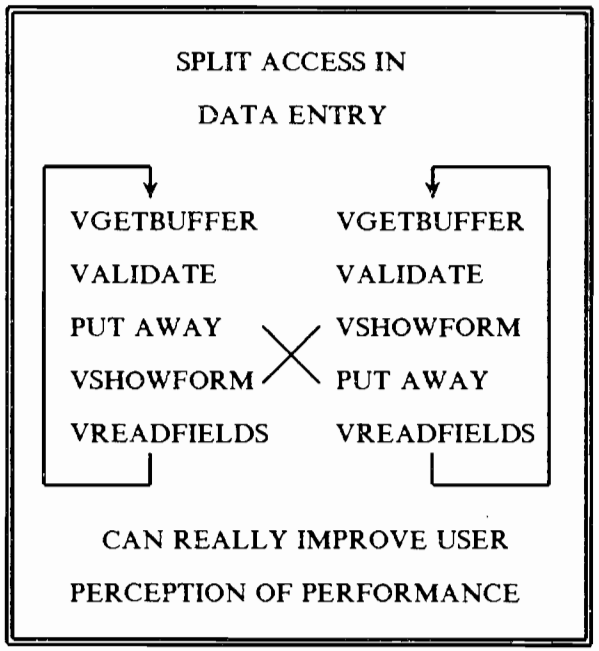

Figure 31

SPLIT ACCESS IN DATA ENTRY (Figure 31)

Performance as perceived by the user is or probably should be our primary concern in transaction processing. This is most critical in data entry applications where uniform response helps set up a work rhythm.

Data entry may involve one or many screens per logical transaction. Multi-screen transactions usually benefit from hardware advances such as the ability to use screens downloaded to the terminal. Single screen transaction performance can be influenced by programming.

Screen data is usually validated and then stored on disc. When the storage is in a data base the overhead for storage may take much time. This time is usually the major contributor to delays in processing functions in data entry.

V/3000 can be used to reduce the delays in data entry. Once you have validated the screen and are certain that only a catastrophic failure could keep you from storing the record, you can return the screen to the user for entry of the next record. While the user fills the next screen, you put away your record.

No data will be lost if the record is not stored before the user hits the enter key or a soft key. These attempted transmissions will be ignored until the program issues a read. In all but a few special cases the validated record will be stored well before the user has submitted the next screen.

This technique will not reduce overhead or performance of individual functions within the program. Responsiveness to the user, however, will be improved. The degree of improvement is proportionate to the amount of work being done parallel to the next transaction think time. An additional benefit is that response will become somewhat less dependant upon overall system load.

PROCESS HANDLING IN DATA INQUIRY (Figure 32)

Application programs often have only the simple capability to allow user data inquiry. These programs must perform well but users expect and usually tolerate reasonable response delays. When the data inquiry is part of a larger function those delays must be minimized.

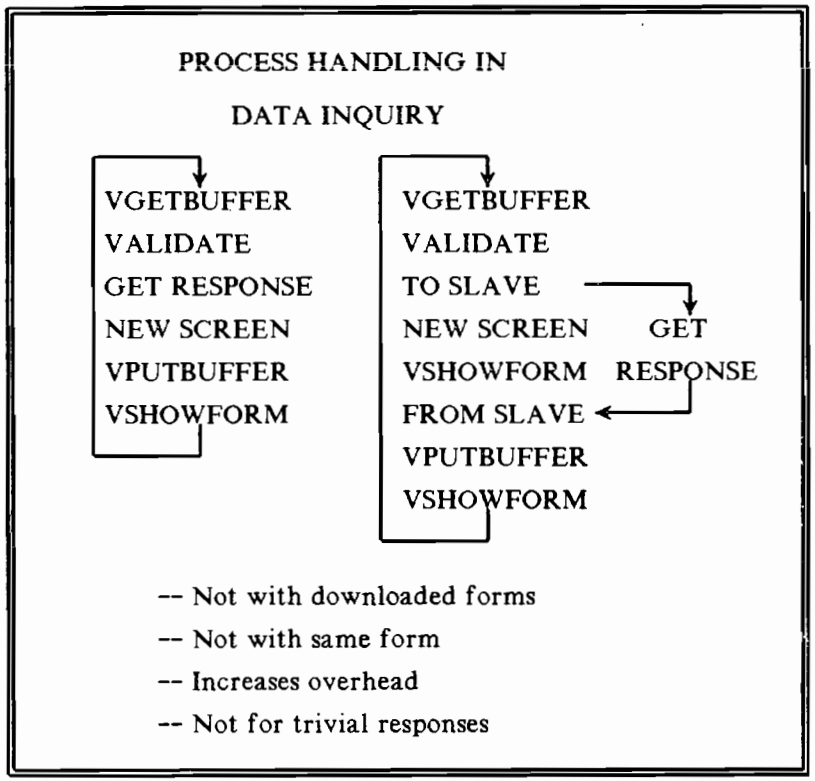

Figure 32

Programs which update existing records have characteristics of both data entry and inquiry. The inquiry part can be a major performance bottleneck. This is quite common since update programs often must retrieve multiple records even though only one may actually be subject to change.

Additional delays are built into such programs if the modifiable response must be displayed on a new screen. User patience is severely tested in many applications because we program serially.

1. We read the request and validate its content. We hit the screen fast if we find errors.

2. We are not so friendly with good input. The user must wait for us to access our data bases to build a reply. Their reward for good input is the chance to watch a cursor blink.

3. We finally respond by sending out a new screen and allow user modification.

4. We satisfy user function but we do it in slow, easy steps.

Process handling can be used to make a program both more responsive and more friendly. We can perform critical time consuming steps in parallel so that delays visible to the program can be invisible to the user.

1. We read requests and handle errors as usual.

2. For good input we immediately send a "please retrieve this data" message to a slave process we have created. The slave was waiting patiently since its only job is to retrieve records from data bases.

3. While the slave gathers records we paint the response screen. This is a friendly act. Our response to good input is probably only a fraction of a second slower than a response to bad input.

4. The faster parallel function waits for the slower to finish. We then retrieve the response buffer from the slave and write it to the screen.

5. Except for a short time needed for process to process communication we have reduced user delay to the delay of the longer parallel process. Even more important, we have become responsive to our users.

This technique is valuable because it improves performance within a technologically limited environment. Changes in technology could make it less valuable or even useless. For example, this technique has no benefit when screens have been downloaded to the terminal since that technology has already eliminated screen painting delays.

We also would not use this technique for trivial responses which take little time to prepare. Process handling absorbs overheads and may not always be cost effective.

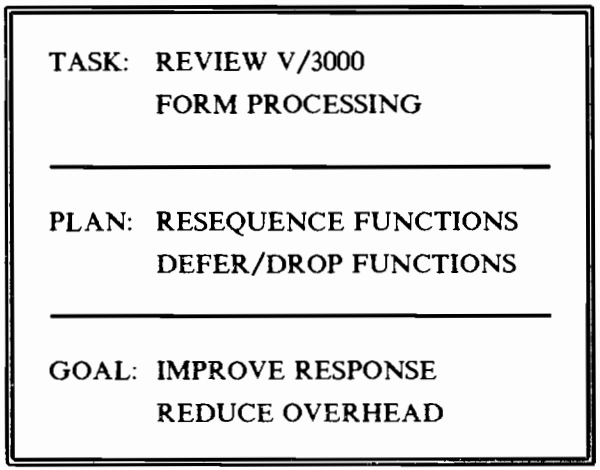

Figure 33

V/3000 FUNCTIONAL SEQUENCING (Figure 33)

Your application defines the V/3000 screens you and your users create for the person to machine interface. Although physical screen design may be critically important to performance it is too broad a subject for this paper. We'll have to assume you've already designed satisfactory screens.

After the screen has been filled, the programmer can begin to manage the processing of the input. Application requirements and V/3000 protocols must be satisfied but the programmer has many options. Some of these options can greatly influence performance.

Application requirements must be met. The programmer should interpret these requirements, however, to see if they can be resequenced more efficiently in the program. Resequencing internal events often changes performance.

V/3000 protocols must also be satisfied. The V/3000 documentation defines a hierarchy among the V/3000 intrinsics and among the functions performed for us by the V/3000 edits. That hierarchy is also open to programmer interpretation. We may be able to improve performance by changing our use of the V/3000 intrinsics.

TASK: Review V/3000 form processing

PLAN: Resequence functions selectively, defer functions when practical, drop functions where possible

GOAL: Improve performance within the program and as perceived by the user.

SOFT KEYS AND SELECTIVE EDITING (Figure 34)

Dumb terminals and unsophisticated terminal I-O limit transaction processing. Intelligent terminals and more sophisticated terminal I-O interfaces remove many of these limitations. We may also have to become more intelligent and sophisticated to more fully utilize our better tools.

Soft keys were among the first improvements as terminals began evolving from absolute dumb to somewhat smart. We take them for granted and use them for standard functions such as "exit" or "refresh". Few of us are using them fully. This paper cannot attempt to cover this topic but we can justify looking at a frequently overlooked usage.

Many applications could flow more smoothly if soft keys could be used to trigger processing functions and if screen information could also be available for processing. A common design technique is to require the user to hit the function defining soft key and then hit "enter" to transmit the buffer. This works but it is neither sophisticated nor friendly.

V/3000 can work with the terminal to allow reading the screen after a soft key. We can trigger this function, called autoread, any time we wish using a simple subprogram or any other

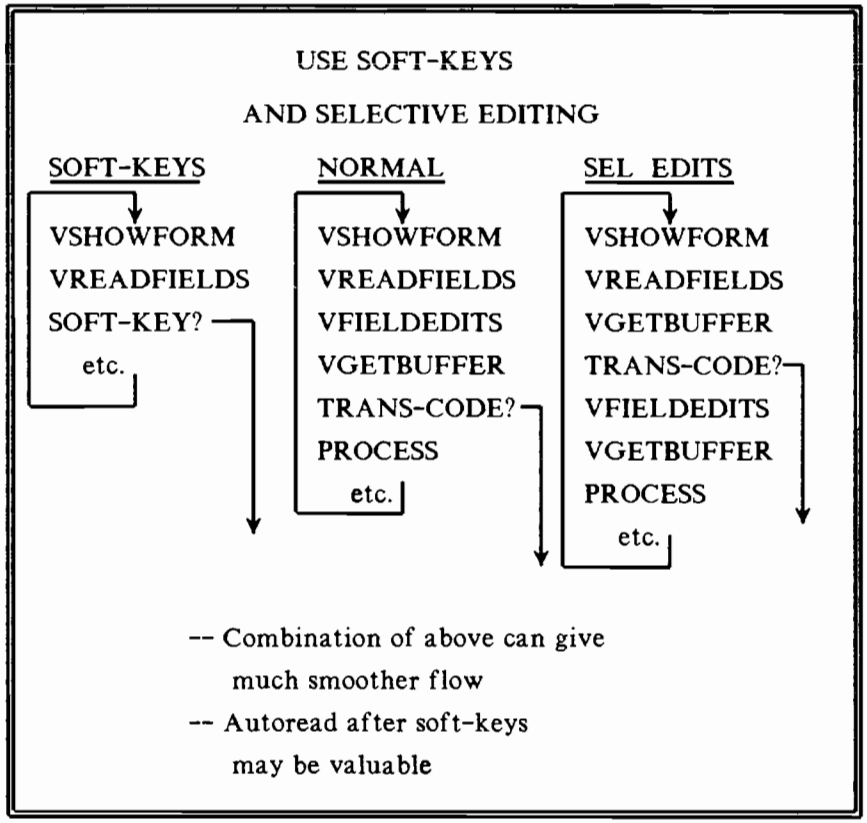

Figure 34

technique to set the autoread bit in the V/3000 common area. Autoread can help make an application run more smoothly and be more friendly. That qualifies as a performance improvement.

Another facility we often ignore is the ability to edit data selectively. V/3000 documentation implies that screen edits must preceed program edits. This is a valid standard for most cases but it may not fit comfortably into all situations. Fortunately, the implied sequence is not mandatory.

When we assume that all screen edits must be done before any program edits we may be painting ourselves into a corner.

Why, for example, should we edit an order quantity when the ordered item is an invalid product. A standard workaround is to avoid V/3000 edits in favor of program edits. Like many workarounds, it works but denies us access to a useful subsystem capability.

We can sometimes "have our cake and eat it too" if we become more flexible. Consider the following sequence of events that uses full V/3000 field edits but allows program intervention at a critical point. This is only a simplified example of sequence editing.

1. Read the screen normally using VSHOWFORM and VREADFIELDS

2. Don't edit the screen yet. Issue a call to VGETBUFFER so you can check a critical field in your program

3. If the critical field fails a test, give the screen back with an appropriate message

4. If the critical field is acceptable, go back to "normal" processing. Issue calls to VFIELDEDITS, etc.

5. You have it made. You get program control at a critical stage and you use V/3000 for less critical work.

DEFER OR DROP PROCESSING (Figure 35)

V/3000 gives us much power with little programming effort. When used sensibly, V/3000 can be good for both the programmer and the user. When used only from the programmers point of view, V/3000 can be a nuisance to the user.

V/3000 gives us three standard processing phases. The initialization phase is relatively little used and causes few problems. We seldom use the edit and finish phases absolutely wrong but we often cause excessive overhead and user aggrevation.

Unless there is some special requirements in an application, some edit functions should never be considered in the edit

DEFER/DROP PROCESSING

DROP	AS FOUND	DEFER
IN 7:204 ...	IN 7:204 ...	IN 7:204 ...
FINISH	JUSTIFY RIGHT	FINISH
JUSTIFY RIGHT	FILL LEADING "0"	JUSTIFY RIGHT
		FILL LEADING "0"
ZERO FILL	IT WORKS !!	WHY NOT WAIT?
NOT ALWAYS		AVOID "JIGGLING"
NEEDED		

Figure 35

phase. Whenever we justify or fill a field in an edit phase statement we are performing a potentially wasted function. Even more important, why should we run the risk of having to rewrite a valid field just because we have altered its form? Users object to excessive screen "jiggling".

We have done only part of our job when we isolate functions within logical V/3000 phases. Whenever possible we should avoid the finish phase until we fully accepted the screen. We waste overhead and risk "jiggling" when we perform this work prior to full screen acceptance.

Some functions can be dropped entirely. Until the machine or operating system are changed, we do not have to fill leading blanks with zeros in numeric fields. The machine treats leading blanks as zeros and allows us to drop the fill function if absolutely necessary for performance.

INTRINSICS AND LANGUAGES (Figure 36)

TASK: REVIEW USE OF
INTRINSICS & LANGUAGES

PLAN: UTILIZE GOOD POINTS
AVOID BAD POINTS

GOAL: IMPROVE PERFORMANCE
AVOID NON-PERFORMANCE

Figure 36

Most programmers prefer a particular language and are proficient in it. In the average application the choice of languages will not determine program performance. We cannot guarantee that any one language is inherently better than another in all possible applications.

We also know that the average application often requires specific functions that may not be available in our language of choice. If we work in a multi-lingual shop we may be able to have a special subprogram written in a language that supplies that function. If our shop is single language we either avoid special functions or devise emulation techniques of some sort.

MPE assists all of us by providing intrinsics for many special functions. Many functions not built into a compiler may be available through an intrinsic. This can simplify our programming and make our jobs much easier.

Somehow we will find ways to get our programs running. All too often we immediately have to find a way to get them running more efficiently. Part of that may require a revaluation of how we have used our languages and the intrinsics.

TASK: Reviews use of languages and intrinsics

PLAN: Use strengths of each and try to avoid weaknesses

GOAL: Improve performance. Of utmost importance, avoid non-performance.

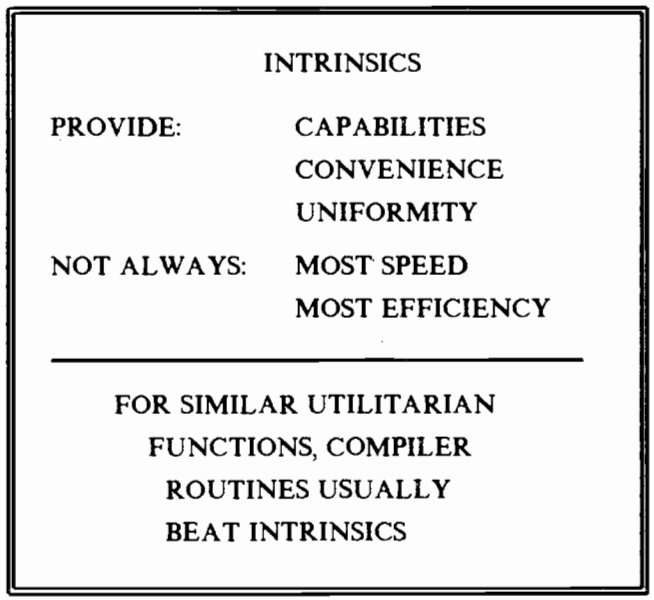

Figure 37

INTRINSICS (Figure 37)

Intrinsics are powerful generalized routines provided with MPE. Some allow limited access to highly specialized functions and others give standardized access to common utility functions.

Intrinsics offer the only access to many functions not available unless you program at the machine instruction level and execute in privileged mode. Access to I-O, for example, is only available through the intrinsics. Compiler library modules invoked by the compilers eventually call intrinsics

for all file access.

All "special capabilities" are available only through the intrinsics. Process handling and data management are invoked using a relatively small subset of the intrinsics. Programmatic communication between computers is handled in similar fashion.

Intrinsics also provide convenient access to many utilitarian functions. Binary to Ascii and Ascii to Binary conversions are common examples. Serial table searches and character transformations are also available.

Specific parameter format and sequence requirements guarantee uniformity. Except for individual compiler conventions and limitations, once you learn to use an intrinsic in one language you should be able to use it in all languages.

SPL comes closest to accessing intrinsics in strict conformance with their documentation. All other languages provide higher level interfaces to one degree or another.

Intrinsics are carefully coded and function efficiently. They are also highly generalized. This generalization leads to relatively high internal overhead. The intrinsic may expend a good portion of its efforts isolating the particular subfunction it is being asked to perform.

Intrinsics will generally be marginally more efficient for file manipulation than the corresponding compiler modules. The difference is slight and seldom justifies the required attention to minute detail. This generalization is meaningless when we use the intrinsics to reach capabilities not available within a particular language.

Compiler routines usually handle discrete data manipulations more efficiently than would the corresponding intrinsic. Even when the compiler routine internally invokes an intrinsic it will not be meaningfully degraded.

Arithmetic functions are particularly well handled by compiler routines. Many of these routines generate highly

efficient machine code. The COBOL compiler uses packed decimal instructions extensively and is surprisingly efficient in handling numeric data.

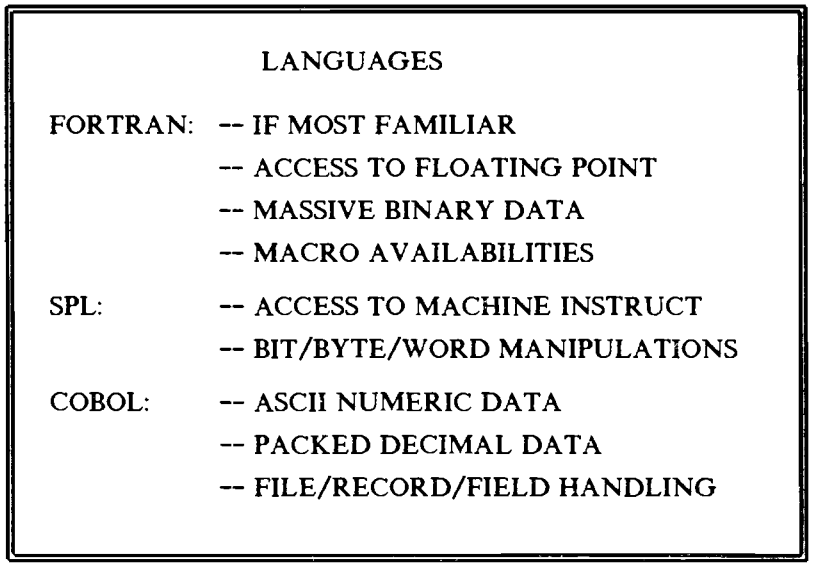

Figure 38

LANGUAGES (Figure 38)

No language is perfect. Each has its special strengths and weaknesses. A language should be evaluated according to how it fits into the application environment.

FORTRAN is an established language with many strong supporters. Programmers either love it or hate it. Either way, there are times when it should probably be your language of choice.

1. If FORTRAN is most familiar to you, why not use it? Most of us usually produce better work when we work with a known quantity.

2. FORTRAN offers excellent high-level access to floating point arithmetic. This is a major strength of FORTRAN and often virtually necessitates its use. FORTRAN is often used to write subprograms accessible from other languages.

3. FORTRAN works very close to the machine when doing binary arithmetic. Massive calculations with binary data often justify FORTRAN for performance reasons.

4. FORTRAN includes many high-level macro constructs. These may make FORTRAN more useful than other languages.

5. Unfortunately, FORTRAN performs poorly with Ascii numeric data and in its current implementation cannot handle packed decimal numerics. Since these data formats are used extensively in commercial applications, FORTRAN is usually a poor choice there.

SPL is the lowest level language on the HP3000. As such, it is potentially the most efficient. It is also relatively tedious compared to most high-level languages.

1. Although SPL is potentially more efficient than any other languages it will probably not improve performance enough to justify wholesale use in commercial applications. Higher level languages are simply better suited for general purpose use.

2. SPL usually fits in best when used for writing specialized subprograms. Some functions are simply not handled well by high-level languages.

3. Machine instructions can be reached in SPL. This is particularly valuable for string and bit manipulations.

4. SPL is most suited for work with low level data. It works especially well at the word level and below. SPL handles records and fields well but the source code tends to become cumbersome and difficult to maintain in large programs.

COBOL is the most widely used high-level commercial language.

Most programmers hate COBOL, some grudingly accept it, and almost none will publicly admit a preference for it. For all its faults it remains the language of choice in most shops.

1. COBOL has a reputation as an inefficient language. COBOL is definitely not the most efficient language but its wide use implies acceptable performance. There is little question that it is highly effective.

2. COBOL is quite efficient in handling numeric Ascii data. Ascii numeric functions are performed using packed decimal arithmetic. Conversions between Ascii and packed decimal data are done efficiently by native machine instructions.

3. Packed decimal data is a standard data format in COBOL under MPE. The ability to deal with packed numerics is a definite advantage for conversions or interfaces with the best known Brand-X computer.

4. COBOL is a high-level language designed to be effective with high-level data. It is most powerful when used to process files, records within files, and fields within records. This is probably the main reason for its wide acceptance.

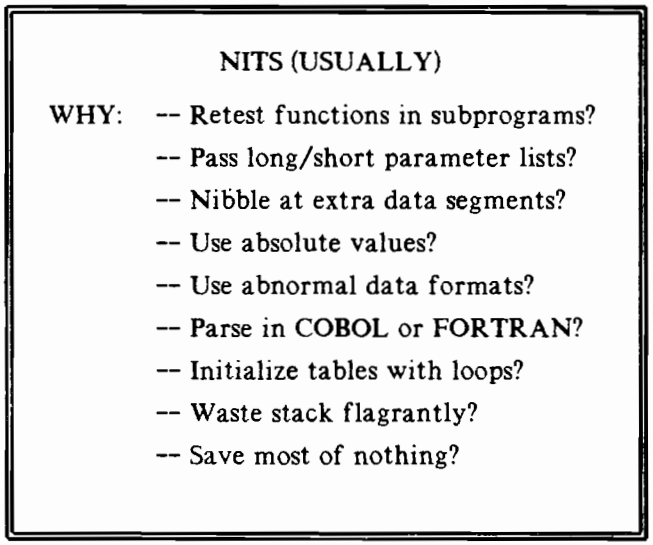

Figure 39

NITS (USUALLY) (Figure 39)

We concern ourselves too much with minor programming considerations. They usually influence performance only slightly. That is not justification, however, for us to code inefficiently.

1. Unless we are writing highly generalized subprograms we probably waste overhead retesting functions. We know why we call a subprogram. In most cases we would be more efficient by writing subprograms with multiple entry points. This avoids retesting to identify our request and also results in more readable programs.

2. Many words have been written about the effect of parameter lists on subprogram efficiency. Long parameter lists are less efficient than short ones. On the other hand, artifically short lists are eventually less efficient if we waste CPU power loading and unloading common control areas. How we enter a subprogram is probably meaningless unless we are doing very little work within the subprogram.

3. Extra data segements have many uses. They can be used quite efficiently but not without added overhead. Whenever data must be moved to or from an extra data segment we should move as much as practical per access. Moving a single word twice costs about the same as moving over 1000 words once.

4. Absolute numeric values are often needed. They cause extra overhead, however, when used in calculations, especially when the result of the calculation is an absolute value. Absolute values force the compiler to generate extra code to guarantee proper results.

5. Packed decimal data has a natural format containing an odd number of numeric digits, each taking up one 4 bit nibble. These plus a 4 bit numeric sign fills complete bytes. If you specify an even number of decimal digits for packed data you force the compiler to do extra work controlling the low order nibble.

6. Character strings are best parsed by specialized routines which utilize special machine instructions. These routines may be designed directly into a compiler or are easily written in SPL. Complex byte manipulations and loop constructs written in highlevel languages are inefficient and should be avoided in most cases.

7. Tables are often initialized using a program loop which

indexes through the table depositing values along the way. This is relatively inefficient and cumbersome. You can save overhead by filling only the first entry and then performing an overlapping move into the rest of the table.

8. Stack space is valuable and should be used with care. Program-directed literals save the stack area needed for valued data elements. But programmers often value documentation over stack. Even so, using a 132 character data element full of spaces to clear a print record is intolerable flagrant waste.

9. Programmers should be efficient but they have to be reasonable first. There is no way we can justify spending time optimizing a small inefficiency when the same effort could be more productive elsewhere. Inefficient coding in insignificant routines does not make inefficient programs.

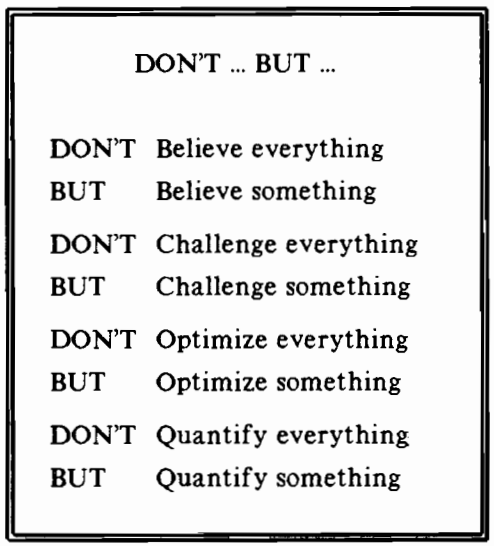

Figure 40

DON'T ... BUT ... (Figure 40)

What was impossible or ridiculous in the past may be standard practice today. What is absolutely true today will quite often be made false by the technology of tomorrow. There are no absolutes but there may be some valuable guidelines.

I'm a programmer, not a philosopher. In Figure 40, I put some comments that have a certain meaning to me. I think they will be more meaningful to you if you supply your own interpretations and meanings.

Good luck and good programming.

SECTION 4 SAMPLE SOURCE LISTINGS

```
* ************************************************************
* THIS IS THE MASTER PROGRAM USED AS A DRIVER TO SHOW THE
*  TECHNIQUES OF HAVING SORTS EXECUTE IN A SLAVE PROGRAM
* THIS PROGRAM COULD HAVE DRIVEN MULTIPLE SLAVE SORT
*  PROGRAMS AT THE SAME TIME IF I WANTED TO TAKE UP THAT
*  MUCH CODE SPACE IN THE HANDOUT
* ************************************************************

$CONTROL USLINIT
 IDENTIFICATION DIVISION.
* DRIVER TO TEST PROGRAM SORTS1
 PROGRAM-ID. FREPSRSS.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. X.
 OBJECT-COMPUTER. Y.
 SPECIAL-NAMES.
     CONDITION-CODE IS COND-CODE.
 INPUT-OUTPUT SECTION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  UNBLOCKER              PIC S9999 COMP.
 01  SAVED                  PIC S9999 COMP.
 01  PROG                   PIC X(10) VALUE "SORTS1R".
 01  PIN                    PIC S9999 COMP.
 01  BFACT                  PIC S9999 COMP VALUE 13.
 01  DSEG-IND               PIC S9999 COMP.
 01  DSEG-ID                PIC S9999 COMP VALUE 43.
 01  DSEG-LEN               PIC S9999 COMP VALUE 79.
 01  REC-AREA.
     05  CURR-COUNT         PIC S9999 COMP VALUE 0.
     05  EACH-REC           PIC X(12) OCCURS 13 TIMES.
 01  DATA-REC.
     05  SEND-COUNT         PIC 999 VALUE 55.
     05  SEND-PROC          PIC 9(9).
 PROCEDURE DIVISION.
```



```
START-ITS SECTION.
START-IT.
    CALL INTRINSIC "GETDSEG" USING
        DSEG-IND DSEG-LEN DSEG-ID
    IF COND-CODE < 0
        CALL INTRINSIC "QUIT" USING 101
    .
    CALL INTRINSIC "CREATE" USING
        PROG \\ PIN
    IF COND-CODE NOT = 0
        CALL INTRINSIC "QUIT" USING 201
    .
    PERFORM SEND-EMS UNTIL
        SEND-COUNT < 1
    MOVE -1 TO CURR-COUNT
    PERFORM ACTUAL-SENDS
    MOVE 0 TO CURR-COUNT
    PERFORM GET-EMS UNTIL
        CURR-COUNT = -1
    DISPLAY "THAT'S ALL, FOLKS"
    STOP RUN
    .
GET-EMS SECTION.
GET-EM.
    CALL INTRINSIC "DMOVIN" USING
        DSEG-IND 0 79 REC-AREA
    IF COND-CODE NOT = 0
        CALL INTRINSIC "QUIT" USING 204
    .
    MOVE 1 TO UNBLOCKER
    PERFORM DOITS UNTIL
        UNBLOCKER > CURR-COUNT
    IF CURR-COUNT NOT = -1
        CALL INTRINSIC "ACTIVATE" USING
            PIN 3
        IF COND-CODE NOT = 0
```

```
            CALL INTRINSIC "QUIT" USING 205
    .
DOITS SECTION.
DOIT.
    DISPLAY "FROM SORT " EACH-REC(UNBLOCKER)
    ADD 1 TO UNBLOCKER
    .
SEND-EMS SECTION.
SEND-EM.
    MOVE 0 TO CURR-COUNT
    PERFORM LOAD-BUFFERS UNTIL
        CURR-COUNT = BFACT OR SEND-COUNT < 1
    PERFORM ACTUAL-SENDS
    .
ACTUAL-SENDS SECTION.
ACTUAL-SEND.
    CALL INTRINSIC "DMOVOUT" USING
        DSEG-IND 0 79 REC-AREA
    IF COND-CODE NOT = 0
        CALL INTRINSIC "QUIT" USING 303
    .
    CALL INTRINSIC "ACTIVATE" USING PIN 3
    IF COND-CODE NOT = 0
        CALL INTRINSIC "QUIT" USING 304
    .
    MOVE 0 TO CURR-COUNT
    .
LOAD-BUFFERS SECTION.
LOAD-BUFFER.
    SUBTRACT 1 FROM SEND-COUNT
    CALL INTRINSIC "PROCTIME" GIVING SEND-PROC
* IF YOU RUN THIS TEST VERSION, DON'T ASSUME COBOL IS
*  SLOW JUST BECAUSE THE CPU TIME SHOWS 4-5 MILLISECOND
*   BETWEEN RECORDS.  THAT TIME PRIMARILY REPRESENTS
*    THE CPU TIME NEEDED TO DO THE DISPLAY STATEMENT
    ADD 1 TO CURR-COUNT
    MOVE DATA-REC TO EACH-REC(CURR-COUNT)
    DISPLAY "TO SORT " DATA-REC

* ************************************************************
* ************************************************************
```



```
* ************************************************************
* THIS IS A SAMPLE OF A SLAVE SORT PROGRAM CONTROLLED BY
*  PROCESS HANDLING TO ALLOW SORTING OF RECORDS OUTSIDE
*  THE MASTER PROGRAM
* MULTIPLE SUCH PROGRAMS CAN BE CONTROLLED AT THE SAME
*  TIME BY THE MASTER
* ************************************************************
$CONTROL USLINIT
 IDENTIFICATION DIVISION.
* KEPT AS SORTS1S, PROGRAM KEPT AS SORTS1R
*  RECEIVES RECORDS FROM SORTMASR
*  SORTS
*  RETURNS SORTED RECORDS TO SORTMASR
 PROGRAM-ID. FREPSRTS.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. X.
 OBJECT-COMPUTER. Y.
 SPECIAL-NAMES.
     CONDITION-CODE IS COND-CODE.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT SORTFILE ASSIGN TO "TEMPSORT,,,,100".
 DATA DIVISION.
 FILE SECTION.
 SD  SORTFILE.
 01  SORTREC.
     05  KEY1              PIC 999.
     05  FILLER            PIC X(7).
     05  KEY2              PIC 99.
 WORKING-STORAGE SECTION.
 01  SORT-FLAG             PIC XX VALUE LOW-VALUES.
 01  TO-WAIT               PIC S9999 COMP VALUE 3.
 01  BFACT                 PIC S9999 COMP VALUE 13.
 01  DSEG-IND              PIC S9999 COMP.
 01  DSEG-ID               PIC S9999 COMP VALUE 43.
 01  DSEG-LEN              PIC S9999 COMP VALUE 79.
 01  REC-AREA.
     05  CURR-COUNT        PIC S9999 COMP VALUE 0.
     05  EACH-REC          PIC X(12) OCCURS 13 TIMES.
 PROCEDURE DIVISION.
 START-ITS SECTION.
```

```
START-IT.
    CALL INTRINSIC "GETDSEG" USING
        DSEG-IND DSEG-LEN DSEG-ID
    IF COND-CODE < 0
        CALL INTRINSIC "QUIT" USING 101
    .
    SORT SORTFILE
        ASCENDING KEY KEY2
        DESCENDING KEY KEY1
        INPUT PROCEDURE GET-INS
        OUTPUT PROCEDURE SEND-BACKS
    STOP RUN
    .
SEND-BACKS SECTION.
SEND-BACK.
    MOVE LOW-VALUES TO SORT-FLAG
    MOVE 0 TO CURR-COUNT
    PERFORM GET-AND-SENDS UNTIL
        SORT-FLAG = HIGH-VALUES
    PERFORM SEND-DATAS
    MOVE 0 TO TO-WAIT
    MOVE -1 TO CURR-COUNT
    PERFORM SEND-DATAS
    .
SEND-DATAS SECTION.
SENDING.
    CALL INTRINSIC "DMOVOUT" USING
        DSEG-IND 0 79 REC-AREA
    IF COND-CODE NOT = 0
        CALL INTRINSIC "QUIT" USING 102
    .
    CALL INTRINSIC "ACTIVATE" USING 0 TO-WAIT
    IF COND-CODE NOT = 0
        CALL INTRINSIC "QUIT" USING 103
    .
    MOVE 0 TO CURR-COUNT
    .
GET-AND-SENDS SECTION.
GET-AND-SEND.
    IF CURR-COUNT NOT < BFACT
        PERFORM SEND-DATAS
    .
```



```
    RETURN SORTFILE AT END
        MOVE HIGH-VALUES TO SORT-FLAG
    .
    IF SORT-FLAG NOT = HIGH-VALUES
        ADD 1 TO CURR-COUNT
        DISPLAY "FROM SORT IN SORTS1R " SORTREC
        MOVE SORTREC TO EACH-REC(CURR-COUNT)
    .
GET-INS SECTION.
GET-IN.
    PERFORM GET-IN-LOOPS UNTIL
        CURR-COUNT = -1
    .
GET-IN-LOOPS SECTION.
GET-IN-LOOP.
    CALL INTRINSIC "DMOVIN" USING
        DSEG-IND 0 79 REC-AREA
    IF COND-CODE NOT = 0
        CALL INTRINSIC "QUIT" USING 111
        STOP RUN
    .
    PERFORM UNLOAD-EMS UNTIL
        CURR-COUNT < 1
    IF CURR-COUNT NOT = -1
        CALL INTRINSIC "ACTIVATE" USING 0 TO-WAIT
        IF COND-CODE NOT = 0
            CALL INTRINSIC "QUIT" USING 105
    .
UNLOAD-EMS SECTION.
UNLOAD-EM.
    RELEASE SORTREC FROM EACH-REC(CURR-COUNT)
    DISPLAY "TO SORT IN SORTS1R " EACH-REC(CURR-COUNT)
    SUBTRACT 1 FROM CURR-COUNT
    .
* ***************************************************************
* ***************************************************************


* ***************************************************************
* THIS IS A DRIVER FOR A SHORT TEST RUN SHOWING SOME OF
*  THE COMPARISONS BETWEEN COBOL AND FORTRAN PERFORMANCE
* IT PASSES ASCII NUMERICS TO A COBOL SUBPROGRAM AND A
```

```
*  FORTRAN SUBPROGRAM
* IT COLLECTS PROCTIMES FOR 1000 CALLS TO THESE SUBPROGRAMS
* THE SUBPROGRAMS EACH ADD 100 6 DIGIT ASCII NUMBERS
*  TO A COUNTER AND THEN RETURN
* THE TIMES ARE REPRESENTATIVE BUT ARE SLIGHTLY LONG
*  SINCE I HAVE NOT BACKED OUT THE TIME FOR THE LOOP
*  CONTROLLING THE 1000 CALLS. THAT TIME IS SMALL
* WHEN THIS WAS TESTED ON A SERIES III IN DECEMBER OF
*  1981, THE RESULTS WERE AS FOLLOWS:
*    COBOL TOOK   16635 CPU MILLISECONDS
*    FORTRAN TOOK 76217 CPU MILLISECONDS
* ************************************************************

$CONTROL SOURCE,USLINIT
 IDENTIFICATION DIVISION.
* KEPT AS FORCOBDA
 PROGRAM-ID. TESTFORD.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  STAMPS                  PIC S9(9) COMP.
 01  STAMPE                  PIC S9(9) COMP.
 01  STAMPD                  PIC 9(6).
 01  THE-SUM                 PIC 9(8).
 01  THE-SUMD                PIC 9(8).
 01  THE-COUNT               PIC 999 VALUE 0.
 01  THE-TABLE.
     05  THE-NUM             PIC 9(6) OCCURS 100 TIMES
         INDEXED BY THE-IND.
 PROCEDURE DIVISION.
 START-OUT.
     DISPLAY "COMPARISON USING 100 6 DIGIT ASCII ENTRIES"
     PERFORM LOAD-EM VARYING THE-IND FROM 1 BY 1
         UNTIL THE-IND > 100
     CALL "COBADD" USING THE-TABLE THE-SUM
     MOVE THE-SUM TO THE-SUMD
     DISPLAY "FROM COBOLII, SUM = " THE-SUMD
     CALL "FORADD" USING @THE-TABLE @THE-SUM
     MOVE THE-SUM TO THE-SUMD
     DISPLAY "FROM FORTRAN, SUM = " THE-SUMD
     CALL INTRINSIC "PROCTIME" GIVING STAMPS
     PERFORM COB-SHOT 1000 TIMES
```



```
     CALL INTRINSIC "PROCTIME" GIVING STAMPE
     COMPUTE STAMPD = STAMPE - STAMPS
     DISPLAY "1000 CALLS TO COBOLII SUMMATION SUBPROGRAM " STA
     CALL INTRINSIC "PROCTIME" GIVING STAMPS
     PERFORM FOR-SHOT 1000 TIMES
     CALL INTRINSIC "PROCTIME" GIVING STAMPE
     COMPUTE STAMPD = STAMPE - STAMPS
     DISPLAY "1000 CALLS TO FORTRAN SUMMATION SUBPROGRAM " STA
     STOP RUN
     .
 LOAD-EM.
     ADD 5 TO THE-COUNT
     MOVE THE-COUNT TO THE-NUM(THE-IND)
     .
 COB-SHOT.
     CALL "COBADD" USING THE-TABLE THE-SUM
     .
 FOR-SHOT.
     CALL "FORADD" USING @THE-TABLE @THE-SUM
     .
* ************************************************************
* ************************************************************


* ************************************************************
* THIS IS THE COBOL SUBPROGRAM MENTIONED IN THE PRECEEDING
*  COBOL MAIN PROGRAM. IT IS AN ASCII NUMBER CRUNCHER
* ************************************************************

$CONTROL SOURCE,SUBPROGRAM
 IDENTIFICATION DIVISION.
* KEPT AS ADDCOBSA
 PROGRAM-ID. COBADD.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  THE-COUNT               PIC S9(9) COMP-3.
 LINKAGE SECTION.
 01  THE-TABLE.
     05  THE-NUM             PIC 9(6) OCCURS 100 TIMES
         INDEXED BY THE-IND.
 01  THE-SUM                 PIC 9(8).
```

```
 PROCEDURE DIVISION USING THE-TABLE THE-SUM.
 START-OUT.
     MOVE 0 TO THE-COUNT
     PERFORM ADD-EM VARYING THE-IND FROM 1 BY 1
         UNTIL THE-IND > 100
     MOVE THE-COUNT TO THE-SUM
     GOBACK
     .
 ADD-EM.
     ADD THE-NUM(THE-IND) TO THE-COUNT
     .
* ****************************************************************
* ****************************************************************


C ****************************************************************
C THIS IS THE FORTRAN SUBPROGRAM MENTIONED IN THE
C  PRECEEDING COBOL MAIN PROGRAM
C ****************************************************************
C
$CONTROL LIST,MAP,LOCATION,STAT
       SUBROUTINE FORADD(INMAT,SUM)
       CHARACTER*6 INMAT(100)
   •   CHARACTER*8 SUM
       INTEGER*4 OUTPUT
       OUTPUT = 0
       DO 60 I=1,100
  60   OUTPUT = OUTPUT + JNUM(INMAT(I))
       SUM = STR(OUTPUT,8)
       RETURN
       END

* ****************************************************************
* ****************************************************************


* ****************************************************************
* THIS IS A DRIVER FOR A SHORT TEST RUN SHOWING SOME OF
*  THE COMPARISONS BETWEEN COBOL AND FORTRAN PERFORMANCE
* IT PASSES BINARY DOUBLE WORDS TO A COBOL SUBPROGRAM
*  AND A FORTRAN SUBPROGRAM
* IT COLLECTS PROCTIMES FOR 1000 CALLS TO THESE SUBPROGRAMS
```



```
* THE SUBPROGRAMS EACH ADD 100 BINARY DOUBLE WORDS
*  TO A COUNTER AND THEN RETURN
* THE TIMES ARE REPRESENTATIVE BUT ARE SLIGHTLY LONG
*  SINCE I HAVE NOT BACKED OUT THE TIME FOR THE LOOP
*  CONTROLLING THE 1000 CALLS. THAT TIME IS SMALL
* WHEN THIS WAS TESTED ON A SERIES III IN DECEMBER OF
*  1981, THE RESULTS WERE AS FOLLOWS:
*    COBOL TOOK   9942 CPU MILLISECONDS
*    FORTRAN TOOK 2499 CPU MILLISECONDS
* ****************************************************************
$CONTROL SOURCE,USLINIT
 IDENTIFICATION DIVISION.
* KEPT AS TESTFORD
 PROGRAM-ID. FORCOBDB.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  STAMPS                  PIC S9(9) COMP.
 01  STAMPE                  PIC S9(9) COMP.
 01  STAMPD                  PIC 9(6).
 01  THE-SUM                 PIC S9(9) COMP.
 01  THE-SUMD                PIC 9(8).
 01  THE-COUNT               PIC 999 VALUE 0.
 01  THE-TABLE.
     05  THE-NUM             PIC S9(9) COMP OCCURS 100 TIMES
         INDEXED BY THE-IND.
 PROCEDURE DIVISION.
 START-OUT.
     DISPLAY "COMPARISON USING 100 DOUBLE ENTRIES"
     PERFORM LOAD-EM VARYING THE-IND FROM 1 BY 1
         UNTIL THE-IND > 100
     CALL "COBADD" USING THE-TABLE THE-SUM
     MOVE THE-SUM TO THE-SUMD
     DISPLAY "FROM COBOLII, SUM = " THE-SUMD
     CALL "FORADD" USING THE-TABLE THE-SUM
     MOVE THE-SUM TO THE-SUMD
     DISPLAY "FROM FORTRAN, SUM = " THE-SUMD
     CALL INTRINSIC "PROCTIME" GIVING STAMPS
     PERFORM COB-SHOT 1000 TIMES
     CALL INTRINSIC "PROCTIME" GIVING STAMPE
     COMPUTE STAMPD = STAMPE - STAMPS
     DISPLAY "1000 CALLS TO COBOLII SUMMATION SUBPROGRAM " STA
```

```
    CALL INTRINSIC "PROCTIME" GIVING STAMPS
    PERFORM FOR-SHOT 1000 TIMES
    CALL INTRINSIC "PROCTIME" GIVING STAMPE
    COMPUTE STAMPD = STAMPE - STAMPS
    DISPLAY "1000 CALLS TO FORTRAN SUMMATION SUBPROGRAM " STA
    STOP RUN
    .
LOAD-EM.
    ADD 5 TO THE-COUNT
    MOVE THE-COUNT TO THE-NUM(THE-IND)
    .
COB-SHOT.
    CALL "COBADD" USING THE-TABLE THE-SUM
    .
FOR-SHOT.
    CALL "FORADD" USING THE-TABLE THE-SUM
    .
* ****************************************************************
* ****************************************************************


* ************************************************************
* THIS IS THE COBOL SUBPROGRAM MENTIONED IN THE
*  PRECEEDING COBOL MAIN PROGRAM
* ************************************************************

$CONTROL SOURCE,SUBPROGRAM
 IDENTIFICATION DIVISION.
* KEPT AS ADDCOBSB
 PROGRAM-ID. COBADD.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  THE-COUNT                PIC S9(9) COMP.
 LINKAGE SECTION.
 01  THE-TABLE.
     05  THE-NUM              PIC S9(9) COMP OCCURS 100 TIMES
         INDEXED BY THE-IND.
 01  THE-SUM                  PIC S9(9) COMP.
 PROCEDURE DIVISION USING THE-TABLE THE-SUM.
 START-OUT.
     MOVE 0 TO THE-COUNT
```



```
     PERFORM ADD-EM VARYING THE-IND FROM 1 BY 1
         UNTIL THE-IND > 100
     MOVE THE-COUNT TO THE-SUM
     GOBACK

 ADD-EM.
     ADD THE-NUM(THE-IND) TO THE-COUNT

* ****************************************************************
* ****************************************************************


C ************************************************************
C THIS IS THE FORTRAN SUBPROGRAM MENTIONED
C  IN THE PRECEEDING COBOL MAIN PROGRAM
C ************************************************************
C
$CONTROL LIST,MAP,LOCATION,STAT
       SUBROUTINE FORADD(INMAT,SUM)
       INTEGER*4 INMAT(100)
       INTEGER*4 SUM
       INTEGER*4 OUTPUT
       OUTPUT = 0
       DO 60 I=1,100
  60   OUTPUT = OUTPUT + (INMAT(I))
       SUM = (OUTPUT)
       RETURN
       END

* ****************************************************************
* ****************************************************************


* ************************************************************
* THIS IS A DRIVER FOR A SHORT TEST RUN SHOWING SOME OF
*  THE COMPARISONS BETWEEN COBOL AND FORTRAN PERFORMANCE
* IT PASSES BINARY DOUBLE WORDS TO A COBOL SUBPROGRAM
*  AND A FORTRAN SUBPROGRAM
* IT COLLECTS PROCTIMES FOR 1000 CALLS TO THESE SUBPROGRAMS
* THE SUBPROGRAMS EACH ADD 100 BINARY DOUBLE WORDS
*  TO A COUNTER AND THEN RETURN
* THE TIMES ARE REPRESENTATIVE BUT ARE SLIGHTLY LONG
```

```
*  SINCE I HAVE NOT BACKED OUT THE TIME FOR THE LOOP
*  CONTROLLING THE 1000 CALLS. THAT TIME IS SMALL
* THE FORTRAN SUBPROGRAM WAS THE SAME AS THE ONE USED
*  IN THE PRECEEDING TEST
* I CHANGED THE COBOL SUBPROGRAM TO SHOW HOW COBOL
*  COULD BE SPEEDED UP USING DIFFERENT CODING
* WHEN THIS WAS TESTED ON A SERIES III IN DECEMBER OF
*  1981, THE RESULTS WERE AS FOLLOWS:
*    COBOL TOOK   6527 CPU MILLISECONDS
*    FORTRAN TOOK 2482 CPU MILLISECONDS
* ************************************************************
$CONTROL SOURCE,USLINIT
 IDENTIFICATION DIVISION.
* KEPT AS TESTFORD
 PROGRAM-ID. FORCOBDB.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  STAMPS                  PIC S9(9) COMP.
 01  STAMPE                  PIC S9(9) COMP.
 01  STAMPD                  PIC 9(6).
 01  THE-SUM                 PIC S9(9) COMP.
 01  THE-SUMD                PIC 9(8).
 01  THE-COUNT               PIC 999 VALUE 0.
 01  THE-TABLE.
     05  THE-NUM             PIC S9(9) COMP OCCURS 100 TIMES
         INDEXED BY THE-IND.
 PROCEDURE DIVISION.
 START-OUT.
     DISPLAY "COMPARISON USING 100 DOUBLE ENTRIES"
     PERFORM LOAD-EM VARYING THE-IND FROM 1 BY 1
         UNTIL THE-IND > 100
     CALL "COBADD" USING THE-TABLE THE-SUM
     MOVE THE-SUM TO THE-SUMD
     DISPLAY "FROM COBOLII, SUM = " THE-SUMD
     CALL "FORADD" USING THE-TABLE THE-SUM
     MOVE THE-SUM TO THE-SUMD
     DISPLAY "FROM FORTRAN, SUM = " THE-SUMD
     CALL INTRINSIC "PROCTIME" GIVING STAMPS
     PERFORM COB-SHOT 1000 TIMES
     CALL INTRINSIC "PROCTIME" GIVING STAMPE
     COMPUTE STAMPD = STAMPE - STAMPS
```



```
     DISPLAY "1000 CALLS TO COBOLII SUMMATION SUBPROGRAM " STA
     CALL INTRINSIC "PROCTIME" GIVING STAMPS
     PERFORM FOR-SHOT 1000 TIMES
     CALL INTRINSIC "PROCTIME" GIVING STAMPE
     COMPUTE STAMPD = STAMPE - STAMPS
     DISPLAY "1000 CALLS TO FORTRAN SUMMATION SUBPROGRAM " STA
     STOP RUN
     .
 LOAD-EM.
     ADD 5 TO THE-COUNT
     MOVE THE-COUNT TO THE-NUM(THE-IND)
     .
 COB-SHOT.
     CALL "COBADD" USING THE-TABLE THE-SUM
     .
 FOR-SHOT.
     CALL "FORADD" USING THE-TABLE THE-SUM
     .
* ************************************************************
* ************************************************************


* ************************************************************
* THIS IS THE IMPROVED COBOL SUBROUTINE MENTIONED IN THE
*  PREVIOUS COBOL MAIN PROGRAM.  IT SHOWS THAT A PROGRAM
*  CODED LOOP CONTROL CAN BE MORE EFFICIENT THAN A
*  COMPILER CONTROLLED LOOP IN SOME CASES
* UNLESS HARD PRESSED FOR PERFORMANCE, I WOULD NOT
*  USUALLY PREFER MY OWN LOOP CONTROL
* ************************************************************

$CONTROL SOURCE,SUBPROGRAM
 IDENTIFICATION DIVISION.
* KEPT AS ADDCOBXB
 PROGRAM-ID. COBADD.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  THE-COUNT               PIC S9(9) COMP.
 01  THE-IND                 PIC S9999 COMP.
 LINKAGE SECTION.
 01  THE-TABLE.
```

```
    05  THE-NUM              PIC S9(9) COMP OCCURS 100 TIMES.
01  THE-SUM                 PIC S9(9) COMP.
PROCEDURE DIVISION USING THE-TABLE THE-SUM.
START-OUT.
    MOVE 0 TO THE-COUNT
    MOVE 1 TO THE-IND
    .
ADD-EM.
    ADD THE-NUM(THE-IND) TO THE-COUNT
    ADD 1 TO THE-IND
    IF THE-IND < 101
        GO TO ADD-EM
    .
    MOVE THE-COUNT TO THE-SUM
    GOBACK
    .
* ***************************************************************
* ***************************************************************


<< THIS SUBPROGRAM ALLOWS BINARY SEARCH OF PB-RELATIVE >>
<<  CODE. IT REQUIRES A FIXED LENGTH ARGUMENT AND      >>
<<  SENDS BACK A FIXED LENGTH RESULT.                  >>

$CONTROL SUBPROGRAM,SEGMENT=FSEARCH
<< KEPT AS SEARCHFS >>
<< FIXED LEN ARG (WORDS) AND FIXED LEN RESULT (WORDS) >>
BEGIN PROCEDURE FINDFIXED(FOUND,ARG,RESULT);
INTEGER FOUND;
INTEGER ARRAY ARG,RESULT;
BEGIN
EQUATE ARG'WLEN = 2;
EQUATE RESULT'WLEN = 17;
EQUATE STEP'SIZE = ARG'WLEN + RESULT'WLEN;
EQUATE ARG'BLEN = ARG'WLEN * 2;
EQUATE RESULT'BLEN = RESULT'WLEN * 2;
EQUATE NUM'COMPARES = 5;
<< FOR NUM'COMPARES:                                  >>
<< IF TOO LARGE, WASTED TIME; IF TOO SMALL, NO-HITS   >>
<< GENERAL GUIDELINE:                                 >>
<< IF NUM'COMPARES ** 2 IS LESS THAN THE NUMBER       >>
<<  OF ARGUMENTS TO BE SEARCHED, THE SEARCH WILL      >>
```



```
<<  NOT SUCCEED IN ALL CASES                          >>
<< IF (NUM'COMPARES -1) ** 2 IS GREATER THAN THE      >>
<<  NUMBER OF ARGUMENTS TO BE SEARCHED, THERE WILL    >>
<<  BE SOME WASTED COMPUTER CYCLES                    >>
INTEGER POS,DISP;
BYTE POINTER RESULTB,BARG;
TOS:=@ARG;
ASSEMBLE (LSL 1);
@BARG:=TOS;
FOUND:="NN";
TOS:=@RESULT;
ASSEMBLE (LSL 1);
@RESULTB:=TOS;
TOS:=1;
ASSEMBLE (LSL NUM'COMPARES);
DISP:=TOS;
POS:=@START'DATA + ((DISP - 1) * STEP'SIZE);
WHILE DISP <> 0 DO
  BEGIN
    DISP:=DISP/2;
    IF POS >= @END'DATA THEN
      POS:=POS - (DISP * STEP'SIZE)
      ELSE
      BEGIN
        TOS:=@BARG;
        TOS:=POS;
        ASSEMBLE (LSL 1);
        TOS:=ARG'BLEN;
        ASSEMBLE (CMPB PB);
        IF = THEN
          BEGIN
            DISP:=0;
            FOUND:="YY";
            TOS:=@RESULT;
            ASSEMBLE (LSL 1);
            TOS:=POS+ARG'WLEN;
            ASSEMBLE (LSL 1);
            TOS:=RESULT'BLEN;
            ASSEMBLE (MVB PB);
          END
          ELSE
          IF < THEN POS:=POS-(DISP * STEP'SIZE)
```

```
          ELSE POS:=POS+(DISP * STEP'SIZE);
      END;
  END;
ASSEMBLE (EXIT 3);
START'DATA:
<< START CONS >>
ASSEMBLE (CON "AB  RESAB                                  ");
ASSEMBLE (CON "BB  RESULT     BB                          ");
ASSEMBLE (CON "CC  RESULT    CC                !!         ");
ASSEMBLE (CON "CEF RES CEF                                ");
ASSEMBLE (CON "CEFARESULT CEFA                          **");
ASSEMBLE (CON "DE  RDE                                    ");
ASSEMBLE (CON "DEF RDEF                                   ");
ASSEMBLE (CON "GHIJRESULT FOR GHIJ----                    ");
ASSEMBLE (CON "GHK RESULT FOR GHK.....................    ");
ASSEMBLE (CON "KKL RESULT FOR KKL                         ");
END'DATA:
END;
END.

* ************************************************************
* ************************************************************


<< THIS SUBPROGRAM ALLOWS BINARY SEARCH OF PB-RELATIVE >>
<<  CODE. IT REQUIRES A FIXED LENGTH ARGUMENT AND      >>
<<  SENDS BACK A FIXED LENGTH RESULT.                  >>
<< THE RESULT IS STORED IN CODE AS A VARIABLE LENGTH   >>
<<  ENTITY AND THE SUBPROGRAM NEED NOT WASTE SPACE     >>
<<  WITH TRAILING BLANKS.  THIS CAN BE SIGNIFICANT.    >>

$CONTROL SUBPROGRAM,SEGMENT=VSEARCH
<< KEPT AS SEARCHVS >>
<< FIXED LEN ARG (WORDS) AND VARIABLE LEN RESULT (WORDS) >>
BEGIN PROCEDURE FINDVARIABLE(FOUND,ARG,RESULT);
INTEGER FOUND;
INTEGER ARRAY ARG,RESULT;
BEGIN
EQUATE ARG'WLEN = 2;
EQUATE RESULT'WLEN = 1;
EQUATE STEP'SIZE = ARG'WLEN + RESULT'WLEN;
EQUATE ARG'BLEN = ARG'WLEN * 2;
```



```
EQUATE RESULT'BLEN = RESULT'WLEN * 2;
EQUATE NUM'COMPARES = 5;
<< FOR NUM'COMPARES:                                    >>
<< IF TOO LARGE, WASTED TIME; IF TOO SMALL, NO-HITS     >>
<< GENERAL GUIDELINE:                                   >>
<< IF NUM'COMPARES ** 2 IS LESS THAN THE NUMBER         >>
<<  OF ARGUMENTS TO BE SEARCHED, THE SEARCH WILL        >>
<<  NOT SUCCEED IN ALL CASES                            >>
<< IF (NUM"COMPARES -1) ** 2 IS GREATER THAN THE        >>
<<  NUMBER OF ARGUMENTS TO BE SEARCHED, THERE WILL      >>
<<  BE SOME WASTED COMPUTER CYCLES                      >>
INTEGER DSTART,DEND,POS,DISP;
BYTE POINTER RESULTB,BARG;
TOS:=@ARG;
ASSEMBLE (LSL 1);
@BARG:=TOS;
FOUND:="NN";
TOS:=@RESULT;
ASSEMBLE (LSL 1);
@RESULTB:=TOS;
TOS:=1;
ASSEMBLE (LSL NUM'COMPARES);
DISP:=TOS;
POS:=@START'DATA + ((DISP - 1) * STEP'SIZE);
WHILE DISP <> 0 DO
  BEGIN
    DISP:=DISP/2;
    IF POS >= @END'DATA THEN
      POS:=POS - (DISP * STEP'SIZE)
      ELSE
      BEGIN
        TOS:=@BARG;
        TOS:=POS;
        ASSEMBLE (LSL 1);
        TOS:=ARG'BLEN;
        ASSEMBLE (CMPB PB);
        IF = THEN
          BEGIN
            DISP:=0;
            FOUND:="YY";
            TOS:=@RESULT;
            ASSEMBLE (LSL 1);
```

```
          TOS:=@DSTART;
          ASSEMBLE (LSL 1);
          TOS:=POS+ARG'WLEN;
          ASSEMBLE (LSL 1);
          TOS:=2;
          ASSEMBLE (MVB PB);
          TOS:=@DEND;
          ASSEMBLE (LSL 1);
          TOS:=POS+ARG'WLEN*2+1;
          ASSEMBLE (LSL 1);
          TOS:=2;
          ASSEMBLE(MVB PB);
          TOS:=@END'DATA+DSTART;
          ASSEMBLE (LSL 1);
          TOS:=2*(DEND-DSTART);
          ASSEMBLE (MVB PB);
        END
        ELSE
        IF < THEN POS:=POS-(DISP * STEP'SIZE)
          ELSE POS:=POS+(DISP * STEP'SIZE);
    END;
  END;
ASSEMBLE (EXIT 3);
START'DATA:
<< START CONS >>
ASSEMBLE (CON "AB  ",0000);     << NO PREVIOUS LEN  >>
ASSEMBLE (CON "BB  ",0003);     << PREVIOUS LEN 03W >>
ASSEMBLE (CON "CC  ",0010);     << PREVIOUS LEN 07W >>
ASSEMBLE (CON "CEF ",0025);     << PREVIOUS LEN 15W >>
ASSEMBLE (CON "CEFA",0029);     << PREVIOUS LEN 04W >>
ASSEMBLE (CON "DE  ",0046);     << PREVIOUS LEN 17W >>
ASSEMBLE (CON "DEF ",0048);     << PREVIOUS LEN 02W >>
ASSEMBLE (CON "GHIJ",0050);     << PREVIOUS LEN 02W >>
ASSEMBLE (CON "GHK ",0060);     << PREVIOUS LEN 10W >>
ASSEMBLE (CON "KKL ",0076);     << PREVIOUS LEN 16W >>
ASSEMBLE (CON -1,-1 ,0083);     << PREVIOUS LEN 07W >>
END'DATA:
ASSEMBLE (CON "RESAB "                          ); << 03W >>
ASSEMBLE (CON "RESULT     BB "                  ); << 07W >>
ASSEMBLE (CON "RESULT    CC              !! "   ); << 15W >>
ASSEMBLE (CON "RES CEF "                        ); << 04W >>
ASSEMBLE (CON "RESULT CEFA                   **"); << 17W >>
```



```
ASSEMBLE (CON "RDE "                                ); << 02W >>
ASSEMBLE (CON "RDEF"                                ); << 02W >>
ASSEMBLE (CON "RESULT FOR GHIJ---- "                ); << 10W >>
ASSEMBLE (CON "RESULT FOR GHK....................." ); << 16W >>
ASSEMBLE (CON "RESULT FOR KKL"                      ); << 07W >>
END;
END.

* ****************************************************************
* ****************************************************************


<< THIS ROUTINE PERFORMS AN AUTOREAD AGAINST A V/3000 >>
<<  FORM. IT IS CALLED WHENEVER YOU WISH TO HAVE THE  >>
<<  SCREEN READ AFTER THE USER HAS HIT A SOFT-KEY AND >>
<<  YOU DO NOT WISH TO FORCE HIM/HER TO HIT "ENTER"   >>
<<  TO TRIGGER THE READ.                              >>
<< IT IS CALLED JUST AS IF IT WERE "VREADFIELDS"      >>
<<  EXCEPT THAT THE NAME WOULD BE CHANGED             >>
<<                                                   >>
<<     CALL "IMMVREADFIELDS" USING VCONT-AREA         >>
<<                                                   >>
<<                                                   >>
$CONTROL SUBPROGRAM
<< KEPT AS IMMREAD >>
BEGIN PROCEDURE IMMVREADFIELDS(CONT);
INTEGER ARRAY CONT;
BEGIN
PROCEDURE VREADFIELDS(C);
INTEGER ARRAY C;
OPTION EXTERNAL;
CONT(55).(13:2):=%1;
VREADFIELDS(CONT);
CONT(55).(13:2):=0;
END;
END.

* 06************************************************************03
* ****************************82****************************
```

JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED

A Critique of *Programming for Performance*

Martin Gorfinkel
LARC Computing
Los Altos, California

The article by Jim May is long—and contains many valuable and thought-provoking suggestions and ideas. It tempts a long response.

The article presents, in clear terms, opinions on a variety of topics relating to programming the HP 3000. If the reader is stimulated to think about his or her own programming practices much will be gained. It is important to publish well-formulated articles of opinion on computing. The reader should be alerted to accept ONLY those portions of the article that seem to be both (a) sensible and (b) applicable to the task at hand. May does a good, but implicit, job of alerting with his last assertion of the introduction.

Languages

FORTRAN and COBOL are the subject of tremendous bias by most programmers. Both languages actually run fairly well on the HP 3000. I agree entirely that the most important consideration is to write in the language that makes you most comfortable.

I find COBOL a very tedious language to use. However, when I had to maintain/improve/update a COBOL program written by someone else—it was not difficult. Similarly, a well-written FORTRAN program does have some structure and is possible to maintain.

SPL has been maligned by Hewlett-Packard for a long time, and May continues the tradition. It is possible to think of SPL as the assembly language of the HP 3000—and to use it in that fashion. It is also possible to use SPL as the ALGOL dialect on the HP 3000 without ever using the low-level features of the language. In its latter form, I find SPL to be the structured language for easy (and non-tedious) programming.

Subprograms

May makes a clear distinction among subprograms, process handling, and standalone separate programs. I would blur that distinction. The important point is: a program must be structured and broken into manageable tasks.

The tools for dividing a programming job into manageable tasks include: SUBROUTINES, PROCEDURES, separate processes, and entirely separate programs. The programmer chooses the appropriate tool depending on the magnitude of the task and the frequency with which the task must be started and stopped.

May advises against using the system SORT program. His example is one in which the stand-alone SORT is called twice (or more) during the processing cycle and a user program operates on the data in the middle. An alternative scenario would have a single call to a SORT either before or after processing by the user program. In that case the convenience and efficiency of the system sort may well outweigh the inefficiency of additional reading and writing of files.

Utilities

The system SORT is in this class, as are V/3000, the IMAGE routines, etc. The point is made that a good programmer could improve on the performance of V/3000 for a specific application. The converse is more often observed: a poor programmer produces less efficient code than is to be found in the system utilities.

An Example

We at LARC have written a system for use with political campaigns, membership organizations, etc. It uses an IMAGE database to keep track of names, addresses, contributions, membership on committees, etc. One program, about 4,400 lines of SPL, handles the data collection with V/3000 screens and produces output reports.

The database contains five manual sets, one automatic set, and eight detail sets. There are twenty-four V/3000 screens used for data collection and report specification.

A tremendous variety of reports is required from the system. It must generate reports on contributions for government agencies, thank-you notes to contributors, mailing labels, letters, rosters of supporters, etc. Report items must be sorted by county, by ZIP code, by city, by last name, etc.

The system could not have been written within reasonable time and cost limits if all of those features had to be written from scratch. The solution was to program a very crude report writer, with no sorting

capability, into the system. The system writes all its reports to disc. A general purpose sorting program, also written at LARC, was used to sort the reports. Printed output is generated by TDP/3000.

Our sort routine is able to sort blocks of records and keep the blocks together—even when the blocks have varying numbers of records. (Thus a list of addresses can be sorted by ZIP code with the multiple lines for one address held together. An address block will contain from three to eight records.)

Our sort utility and TDP were working well prior to the development of this system, thus reporting required minimal debugging. There is some loss of efficiency as a report file is written, closed, opened by the sort routine, closed again, then re-opened by TDP. That loss is small compared to the advantages gained in programming time, lack of debugging problems, and reporting generality, achieved by using the previously developed tools.

Optimizing

Any program that is going to be used repeatedly is a candidate for review/rewriting/optimization. Three questions should be addressed in planning that review: (1) who does it; (2) how often is it done; and (3) what do you look at.

Ideally, a programmer other than the author of the program is assigned to the review. In programming, as in writing text, it is very much easier to find the errors and the places for improvement in someone else's work. A side advantage for the installation is that two programmers, rather than one, are familiar with the details of the code after a review.

With large programs under continuing development, the review cycles should alternate with enhancement cycles. If version "n" of a program contains new features and bug fixes, then version "n + 1" should be a clean-up version with improved code and no new features.

May covers what to look at as a program is reviewed.

Futures

The good programmer must consider the future of the program. That consideration takes many forms: structure the code for easy maintenance; anticipate future demands and provide the hooks for those in your code; try to make the code understandable for the person who will have to maintain (enhance) it; and DO NOT rely on clever undocumented features of the HP 3000.

Closing

My main interest is how well a program runs rather than how fast. It is more important to add the nuances of user convenience than to speed up processing by 5%. (There must be a balance between function and speed. A perfectly functioning program that takes two days to process one day's worth of input is useless!)

The rule for getting programs that function conveniently and in a user-friendly mode is: Have the author (programmer) work with the program in production mode.

If the program is to be used by data clerks, then the programmer (a) plays data clerk and enters data; and (b) works with the other data clerks and hears their complaints/compliments.

There are two objectives, equally important:

- The designer/programmer should hear the complaints and see the inconveniences in the program so as to be able to improve it.
- The designer/programmer should hear the compliments that will come from a job well done!

Someone who never has direct contact with a program in production mode never finds out how convenient it is to use. Features that might be easy to implement and very important to the user are never noticed—thus never implemented.

If the programmer gets bug reports and complaints through a filter and without any direct contact with users, the compliments do not get through. The programmer gets no positive feed back on the product. The result is a programmer burned out and unwilling to deal with users.

JOURNAL
OF THE HP 3000 INTERNATIONAL USERS GROUP, INCORPORATED

A Critique of *Programming for Performance*

Ross Scroggs
The Type Ahead Engine Company
534 Rosal Avenue
Oakland, California 94610
415/835-5603

Very few HP 3000 users have unlimited computing power that they can use without regard to cost: They must extract the maximum performance from their machines. Unfortunately, there are very few HP-supplied references that describe how to maximize performance. Programming for Performance (PFP) is one of the few independently produced papers that attempts to fill the gap of missing information. (Though the author is an HP employee, the paper represents a personal effort and is not official HP information.) PFP should be read as a high-level guide to areas of programming on the HP where more thought and information will yield increased performance. It is not a low-level, nuts-and-bolts, discussion of specific topics. There are other papers that address these issues, so all material should be read to obtain the complete performance picture. (The best source for performance papers is the proceedings of the 1981 IUG meeting in Orlando; the proceedings of the 1980 IUG meeting in San Jose are also useful.)

PFP makes a strong case for programming with subprograms: This is the most important issue in the paper. The technique simplifies design, coding, and testing. The other papers should be consulted for specific information on how to instrument subprograms to identify the one that bears the most performance improvement work.

NOBUF file processing is gaining popularity as a method to improve sequential access. The overview presented does not address the specific problems associated with NOBUF I/O and in fact may give misleading ideas about blocking factors. A problem that isn't discussed is how to access a file that is used both sequentially and randomly. Using small blocking factors with buffered I/O makes random-access programs simple and efficient; using MR/NOBUF access to read multiple small blocks in sequential access makes batch programs efficient. You get the best of both worlds, but beware of the associated problems.

The section on sorting introduces an idea that many hard-core HP programmers use—process handling. It is an under-used technique, and the sorting example is a good illustration of the benefits to be gained from the method.

Maintaining small tables of data in SPL procedures that use PB arrays is a good idea; it allows fast access to the data and is good modular design. Maintaining the data in any form other than the source is unacceptable; changing a data structure by directly updating a USL file makes me shudder. The SPL program given to illustrate the technique is a poor example of the use of this language; look elsewhere for a sample SPL binary-search routine.

The section on V/3000 presents some simple but effective ideas on improving the response time to the user. The idea of painting the next screen after applying the edits to the current data, but before updating the database, allows the user the opportunity to input data while the HP is processing. It presents a user-education problem though; someone must explain to the user that the ENTER key is sometimes inoperable. Eveyrone's life will be considerably complicated if users start to bang the ENTER key several times for each transaction to make sure that the computer received it. It should also be pointed out that character mode with formatted screens may, in fact, give the user better performance than block mode. There is more immediate response to errors and special conditions; the user doesn't waste time filling in the entire screen to discover that the transaction is invalid. At the expense of some increased processing, the user can be guided through the form, filling in only the required fields, and the program can skip the fields that are unnecessary for this transaction. There is a corresponding savings—all those empty fields are not transmitted to the computer.

Choosing the right language for a programming project is important; and PFP says just that—choose the right language. Number crunching in COBOL just doesn't make sense; FORTRAN has it wired. It should also be pointed out that the new transaction and report-writing tools are now an important alternative to any existing programming language. Use the right tool for the job: Learn a new language for that new 10,000 line project; don't use BASIC just because you know it.

A performance tool that got little mention was segmentation. Segmenting a program correctly is fairly easy, but it won't dramatically improve performance. A poorly segmented program, though, doesn't perform well. Spend a few minutes with segmentation; it doesn't hurt.

I recommend this paper to people concerned with getting maximum performamce from their HP, but consult some of the other reference material available for some of the low-level tips and techniques that will help implement the ideas presented here. I hope that PFP will inspire other HP people to write down their ideas; there is no end to the need for information of this type.

Journal
Administrative Offices
HP 3000 International Users Group, Inc.
289 South San Antonio Road
Los Altos, California 94022
U.S.A.