System Performance and Optimization Techniques for the HP3000

John E. Hulme
President of Applied Cybernetics, Inc.

INTRODUCTION

The purpose of this paper is to introduce the reader to certain techniques which can improve system performance, throughput, and run-time efficiency on HP3000 computers. These improvements will typically reduce response time substantially and generally increase data processing productivity.

This paper will not simply tell you what to do and what not to do. In many cases there are trade-offs involved and it is more important to understand the principles behind the techniques than the techniques themselves. And because analogies often help us to learn by giving us a new perspective, we will make use of a non-data-processing illustration.

SOME BASIC PRINCIPLES

The first thing to understand is that any given computer can execute a finite number of instructions in a fixed amount of time. When that theoretical limit is reached, no amount of tuning can "squeeze" extra instructions into the computer. For the most part, however, computers do not bog down because we ask them to do too much, but rather because we cause them to trip over themselves in the process of doing it.

This leads to the second important principle: At any moment the computer is either (1) doing productive work; (2) getting ready to do productive work; or (3) waiting on some external action before it can proceed with productive work. As a program is initiated, thereby causing a certain sequence of instructions to be executed, we will call the execution of those instructions "productive work." Whether the "productive work" is really necessary or not, and whether it is efficiently or inefficiently organized, are issues to be addressed later. But a more significant fact of computer life is that usually only a small percentage of the computer's time is spent executing application program instructions.

A CRUDE MODEL

To illustrate these principles, imagine a "library for the blind." The librarian sits behind the desk waiting for a blind person to walk into the library. This is the "waiting period." When the blind person arrives, the "getting ready" period begins. The blind person tells the librarian which book to retrieve and by one method or another the book is retrieved. The librarian now begins the "productive work" phase, reading to the blind person from the selected book. When the reading is completed, the librarian may return the book to the shelf or leave it on the desk. Then a new waiting period begins.

If the library is a busy one, we can imagine that one or more assistants might be hired to transport the books between the librarian's desk and the book shelves. Let's imagine that there is one assistant for each wing of the library. The librarian can do more *productive work* (reading to the patrons), spending less time *getting ready* (still look things up in the card catalog, but now dealing with the assistants instead of transporting books). A new type of waiting is introduced, however: waiting for assistants to bring books back.

In this analogy, the librarian represents the computer's central processing unit (CPU), by which all the productive work is accomplished. Like our imaginary library, the HP3000 has only one CPU. To improve throughput we must maximize the CPU's productive time.

Each patron represents a log-on session or job. The librarian's desk represents the computer's main memory. It is of a limited size, merely a workspace, in comparison to the stacks of book shelves which correspond to the mass storage devices. Finally, each assistant represents an I/O channel transferring data to and from disc, for example.

While illustrating some important concepts, this analogy does not accurately model the run-time environment of the HP3000, or any other computer. How could we refine the model to make it more realistic?

THE MODEL REFINED

At the risk of distorting the human situation, let me suggest four refinements which make our model more nearly resemble the actual computer processes:

1. The "library" should be regarded as a collection of (a) read-only instruction manuals and reference tables (programs and constants) and (b) numerous loose leaf volumes (files) containing sheets of current figures and data (records) which may be periodically replaced, revised, removed, or added to.

2. The 'librarian's" job should be generalized to include any type of service that can be performed on the

basis of preprinted instructions and supplied data.

3. The computer always deals with a *copy* of whatever is stored on the disc, and usually just a few records at a time. So let's imagine that instead of asking a library assistant to fetch a particular book, the librarian will specify a limited number of paragraphs or data sheets and will ask the assistant to bring a photocopy of the desired paragraphs (colored paper for instructions; white paper for data).

4. Because the processing speeds of a computer are so great, our model operates in slow-motion by comparison. Allowing that the librarian can do in one hour what an HP3000 can do in one second (i.e., using the scale of one hour for each second), the assistant could handle 20 to 60 requests per hour, and the equivalent of a 60-word-per-minute typist could enter one character every 12 minutes. A 2400-baud rate would be equivalent to a maximum of 5 characters transmitted per minute, and a 600-line-per-minute printer would correspond to one line every 6 minutes.

SLOW MOTION PERFORMANCE SIMULATION

Visualize this scenario from the patron's point of view (refer to Figure 1): You walk into the library, find an empty cubicle (terminal), and make yourself comfortable. You begin to formulate and transmit your library card number and password (log-on) at the rate of no more than 5 characters per hour. (If it will relieve the agony, you may imagine that you spend the time drawing very large, very elaborate block letters). Depending on the facilities available in the cubicle, you will either transmit each letter as it is formulated or accumulate several characters (maybe even hundreds) and transmit them in a burst. In either case, you transmit each letter separately by ringing a bell, and, when you have the librarian's attention, holding up the card with the letter on it. The librarian records each character of your message on a notepad corresponding to your cubicle, then continues with his other business. Finally you send a character which means "that's the end of what I'm sending you."

The librarian eventually verifies that you are a qualified user of the library and sends you back a standard message which allows you to proceed. This process may require the librarian to send his assistant to the book shelves several times, e.g., to get a procedures manual, index of users, table of passwords, welcome message, etc.

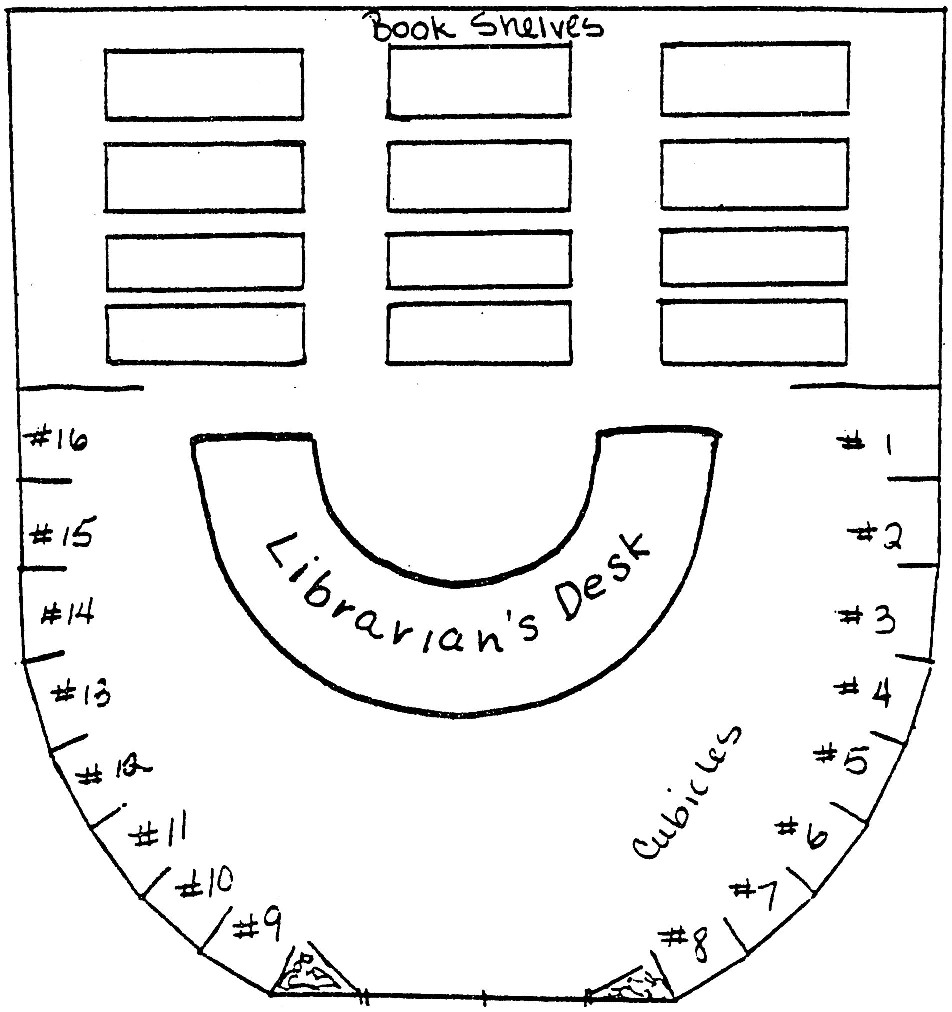

Figure 1. The Library

Next, you painstakingly tell the librarian the name of an instruction manual (program) you want him to follow in performing some service for you. He has the assistant get him a copy of the first paragraph (segment) of the instruction manual (unless a copy happens to be sitting somewhere on the desk already). He also gets a copy, your own personal copy, of a worksheet (your data stack) associated with the specific instruction manual you have specified.

In case there is not enough empty space on the desk for these papers, the librarian first clears some space by either (a) throwing away one of the instruction sheets, (b) having his assistant put the worksheet for some other patron in a special holding file (virtual memory), or (c) having his assistant take one of the data sheets back to the loose-leaf it was copied from and replace the original with the new version.

The librarian now goes to work following the instructions you have requested. This will continue until (a) he comes to a point in the instructions which specifies he is to send certain information to you and/or ask you for additional input; (b) he comes to the end of the page or is otherwise instructed to refer to another page, one which is not currently on the desk; (c) the instructions require that information be fetched from the book shelves, taken there to be filed, or sent to some output device; (d) a predefined length of time elapses (a 500 microsecond quantum corresponds to one-half hour in our model); or (e) the librarian completes his assignment and disposes of your worksheet.

In any of these cases, the librarian will go back to work for one of the other patrons, provided he has all the resources necessary to do so. If not, he will wait (until the necessary information is fetched by the assistant or transmitted by one of the patrons). Depending on what you've asked the librarian to do, and how busy he is doing things for the other patrons, it may take hours or even days before he gets back to you. But then again, it may take days for you to formulate the equivalent of one screen of input, too (at the rate of 5 characters per hour).

THROUGH THE EYES OF THE CPU

Now let's reverse roles and look at the situation from the librarian's perspective. Try to imagine yourself as a calm, unemotional, purely methodical being who is never responsible for mistakes because he does precisely as he is told. You couldn't care less if someone gets poor response time; you aren't to blame, because you only rest when there's nothing for you to do. In fact, you purposely set things aside during peak demand periods to do in your spare time. But you can't take credit for that either — you're only following directions from the MPE handbook.

2:08:17 Ring! There's the bell in cubicle five. He's holding up the letter "R." Write it down on memo pad #5 (line buffer).

2:08:20 Here's the library assistant with the record session #12 requested. Oops! The worksheet for session #12 has been set aside (swapped out to the system disc). Send the assistant for it and wait a minute.

2:08:24 A ring from cubicle #8. That's a carriage return. Time to reinitiate session #8. Make a note to send the assistant for the worksheet when he gets back.

2:08:29 Wait some.

2:09:00 Wait some more.

2:09:16 Oh good, something to do (the observer's feelings, not yours). A ring from cubicle #3. A "7". Write it down.

2:09:20 Here's the assistant. Put worksheet #12 on the desk. Send him back for worksheet #8 — no, there's not room for it. Give him the worksheet for session #5 and send him to file it (we're waiting for input from cubicle #5). We'll send him for worksheet #8 next time.

2:09:24 Okay, now to get to work on task #12. First set the timer for 30 minutes. Now add I to J and put the result in K.

2:09:28 Move W6 to W2. Move . . . hold it, there's another ring from #3. Say, that's only a few seconds . . . must be a block-mode terminal. Write down the "9" and go back to work. Move X to Y. Call the procedure "XFORM." Oh, it's on the desk already — it hardly ever gets thrown out, that's because nearly every program uses it.

2:09:40 Another ring from cubicle #3. This time it's a minus sign. Continue with "XFORM." Convert the first letter of Y to upper-case. Now the second letter. Now the third. Now the fourth. That's all. Return to the main program. It's still in memory. Move the new Y to F3.

2:09:52 Another ring from cubicle #3. A field separator. Resume task #12. Perform FLAG-SET subroutine. It's in another segment, one that's not in memory. Make a note to send for it. Suspend task #12 for a minute.

2:10:04 Cubicle #3 again. Just a blank, but write it down anyway. That's "7-9-minus-field separator-space" so far.

2:10:14 The assistant has finished filing worksheet #5. Send him now for worksheet #8.

2:10:16 Cubicle #3. Another space.

2:10:19 Interrupt from the printer saying the last line has printed successfully. Now reactivate the spooler job — it's instructions are still on the desk and so is the buffer containing the print-line. Initiate I/O transfer.

2:10:26 2-second wait.

2:10:28 Cubicle #3. A third space.
12-second wait.

2:10:40 Cubicle #3. A fourth space. 12-second wait.

2:10:52 Cubicle #3. A fifth space. 12-second wait.

2:11:04 Cubicle #3. A field-separator. 5-second wait.

2:11:09 Worksheet #8 is here. Send assistant to get a copy of FLAG-SET routine. Now to process this input from cubicle #6.
Edit first field. OK. Edit second field. OK. Move first field to R1.

2:11:16 Cubicle #3. The letter "H".
Move second field to K2. Edit third field. Isn't numeric but should be. Transfer to error handler in same segment.

2:11:28 Cubicle #3. The letter "O".
Prepare output to tell cubicle #8 about error. Comment: It's a shame, but since he's in block-mode, he'll have to retransmit the whole screen again, after correcting the error in field 3. And who is to say other errors might not be detected after that? And you, the librarian, can receive those 873 characters, one every 12 seconds for nearly three hours. But you don't mind. It's only a job.

2:11:40 Cubicle #3. The letter "V".
Finish putting error message in the output buffer. Initiate transfer to cubicle #8. Mark task #8 eligible to be swapped out.

2:11:47 Cubicle #11. The letter "P".

2:11:52 Cubicle #3. The letter "E".
FLAG-SET routine is here. Continue with task #12. Move 1 to FLAG. Add 1 to COUNT. Exit back to mainline. What! The assistant had to fetch a separate segment just so we could do that?

2:11:59 Cubicle #11. Oh, oh. Two block-mode devices transmitting at once! Record the letter "I".

2:12:04 Cubicle #3. The letter "R".

Comment Stop, I've had enough of dinging bells! This place sounds like a hotel lobby, not a library!

OBSERVATIONS

As this analogy indicates, there are three factors which reduce overall system performance:

1. Unnecessary disc I/O (most serious);
2. Unnecessary terminal I/O (too common); and
3. Unnecessary CPU usage (rarely the problem in an on-line environment.

Excessive Disc I/O

The primary cause of excessive disc I/O is *inadequate main memory* to hold the required work space (stack and data segments) for each concurrent process, plus all frequently referenced program segments, plus a reasonable mix of infrequently referenced program segments.

The HP3000 is very good at handling multiple concurrent users, even when they won't all fit in memory together. In fact, the use of virtual memory, combined with a well-designed algorithm for selecting which segment to overlay, allows the system to operate efficiently even in cases where a single program exceeds the limits of main memory.

The thing to remember, however, is that code segments put a relatively small load on the system while data segments put a potentially disastrous load on the system. In the first place, code segments can be split up and made as small as the programmer wants them to be. Secondly, they do not have to be rewritten to virtual memory when the main memory space is to be re-used; they are simply overlaid. Data segments, on the other hand, tend to expand, and can be split only with difficulty. Since their contents may change, they must be rewritten each time the process is swapped out, and reread each time it is swapped back in. Finally, whatever data space is required must be repeated for each process that is active. Therefore, if you are supporting 20 terminals, any reduction in data requirements would produce 40 times the benefit that an equivalent reduction in code requirements would produce.

Aside from upgrading to a larger machine, a shortage of main memory can be averted by:

1. Reducing the number of concurrent processes (not an attractive option);
2. Reducing the average stack or data segment size;
3. Reducing the size of the average program segment;
4. Organizing program segments better so that out-of-segment transfers occur less often to non-resident segments and so that often-used code is collected in compact segments that are likely to stay in memory, or
5. Some combination of the above.

When adequate main memory is available, swapping is unnecessary, and disc accesses (which are very expensive in terms of time) will be expended strictly for data retrieval and storage. Once swapping begins, the computer's "productive" activities are at the mercy of "waiting." In the worst case, "threshing" occurs, which means that every time a session gets a turn at execution, either the program segment has been overlaid or the session's work space has been swapped out.

It is worth noting that the use of IMAGE (or of KSAM) causes the allocation of extra data segments. Specifically, each IMAGE database that is open requires a data segment large enough to hold one copy of the root file plus four complete database buffers. If a program accesses multiple databases, or if the root file or buffers are large, the memory requirements will be substantial, and with many terminals running database applications, the memory requirements can add up very quickly. Granted, the advantages of using a powerful access method may outweigh the costs of additional memory demands, but such tools should be used carefully and not indiscriminantly.

It should also be noted that the use of block-mode requires extensive buffers in the stack (at least as large as the largest screen to be transmitted). The use of

VIEW/3000 may add another 6000 bytes of buffer in each user's stack, not to mention the extra data segments created by its use of KSAM. If you have 20 users, this amounts to 120K extra bytes of memory or more.

Excessive Terminal I/O

Major causes of excessive terminal I/O include the following:

1. Transmitting unnecessary characters (trailing spaces, leading zeroes, insignificant digits, etc.) to the computer, a necessary consequence of fixed-format or block-mode input.
2. Transmitting the same data to the computer more than once, as occurs in block-mode when a whole screen is retransmitted to correct an error in a single field.
3. Retransmitting to the computer data which has not been changed since it was received from the computer. This too is the result of block-mode transmission.
4. Repeatedly displaying prompts at the terminal instead of using protected background forms.

Since each character of input consumes critical resources, every effort should be made to ensure that only significant data is transmitted (no extraneous zeroes or spaces and only those fields that are new or have been modified).

It is not only wasteful of computer power, but also destructive of operator morale, to wait until a whole screen of data has been entered and transmitted to the computer before discovering that the screen is invalid due to a duplicate key or an unrecognized search-item value, etc.

It is equally inefficient (for the computer, that is) to display a screen of data, have the operator update a single value and transmit the whole screen back to the computer. In an extreme case, this could amount to over a thousand characters transmitted just to change one or two characters.

Excessive CPU Usage

Besides the costly I/O overhead, it is altogether possible that a retransmitted screen will be completely re-edited, values packed and unpacked, and fields reformatted even though only a single field was updated, and maybe even if *nothing* was updated. This is one cause of unnecessary CPU usage.

Most editing and reformatting done in COBOL subroutines requires excess usage to begin with, and it is far better to allow such work to be done in SPL subroutines, where it can be done efficiently. Including such subroutines in the COBOL programs also causes bulkier segments, which is likely to increase the need for swapping. The best solution is to incorporate all editing within the terminal-handling module itself, since it is already being shared by all on-line programs and is therefore likely to remain constantly in main memory.

There are a multitude of factors which can unnecessarily increase the so-called "productive work" which the CPU has to do. Because computers are seldom CPU-bound in an on-line environment, few people exert the effort to truly optimize CPU performance anymore. Whenever it is a problem, more careful analysis of the program(s) in question will usually yield a more efficient method of solving the application problem.

Often, more careful analysis will also yield a better solution from the point of view of disc I/O as well, both in terms of swapping, code-segment switching, and data retrieval and storage. One word of warning, however: more efficient solutions (CPU-wise) are very often more complex, and to the extent that they increase stack space, or code-segment size, or they require more transfers from one code-segment to another, they may prove counter-productive.

One situation in which heavy CPU usage can be very detrimental is when on-line processes are competing with batch applications for CPU resources. This can be vividly illustrated by running a COBOL compile, an Editor GATHER ALL, a sort, or the BASIC interpreter at the same time on-line programs are running. Block-mode applications exhibit many of these same tendencies and can severely impede response time for character-mode applications when both types are running concurrently.

SPECIFIC OPTIMIZATION TECHNIQUES

1. Resegment programs so that no segment exceeds %5000 words.
2. Set the blockmax parameter on IMAGE schemas as low as possible.
3. Use extra data segments where possible and free them up when finished, rather than increasing stack space for large temporary buffers.
4. Don't keep files open unnecessarily.
5. Don't abuse IMAGE:
   a. eliminate sorted chains where possible.
   b. carefully evaluate tradeoffs of increasing or eliinating secondary paths in detail data sets.
   c. use "@;" or at least "*;" for item lists wherever possible.
   d. only use binary keys (in master file) when overlapping keys can be avoided.
   e. don't let synonym chains get very long.
   f. when loading master data sets, store only primaries on the first pass, makng a second pass for secondaries.
   g. keep master data sets less than 85% filled.
   h. periodically reorganize detail data sets that have long chains associated with a frequently-accessed path (puts consecutive records in the same physical block).
   i. keep the number of data sets in a database as small as practical without requiring many programs to open multiple databases.
   j. keep IMAGE record lengths to a minimum.

6. Have operators exit programs when not in use.
7. Use a field-oriented terminal handler which performs standard edits for you.
8. Use formatted screens with protected background whenever the application is appropriate to such use.
9. Keep terminal I/O buffers small; if possible, eliminate block-mode I/O altogether. (Don't use block-mode and character-mode I/O at the same time.)
10. Don't use VIEW without a lot of memory.
11. Don't use DEL at all.
12. Run CPU-intensive jobs (including compiles, preps, and Editor GATHER ALL) when on-line applications are not running, or at least run them in a lower-priority subqueue.
13. Set the system quantum for a shorter priod than recommended in the MPE manual (but don't overdo it — some experimentation may be necessary).