Dynamic management of free storage in a time-sharing operating system was studied empirically by the techniques of monitoring, emulation, and on-line experimentation.

A new algorithm, based on observed usage patterns of different block sizes, was implemented and evaluated. On-line experiments demonstrated that supervisor time spent in free-storage management was reduced by seven or eight to one.

Analysis of free-storage algorithms

by B. H. Margolin, R. P. Parmelee, and M. Schatzoff

Central to successful operation of a computer system based heavily on reentrant (or pure) procedures is reliable and efficient dynamic management of free storage. Such systems must allocate, use, and release one or more blocks of free storage for each task or system operation, e.g., each I/O task or each request for supervisor services. For any list-processing system, such as AED or LISP, efficient management of free storage is a fundamental problem. The consequence of errors in allocation and release is usually total collapse of the system; that of mismanagement is usually processor inefficiency or under-utilization of the free-storage pool. The processor inefficiency resulting from a poor or ill-chosen management algorithm is usually tolerated, even though it is high relative to that of other system functions; under-utilization of free storage is less tolerable, as "lock-up" can be encountered. This circumstance must be averted by task deferral or "garbage collection" procedures, both of which are costly to system performance.

To date, discussions of free-storage management have treated either: (1) a specific algorithm, which is developed and supported post-hoc, e.g., on the basis of a simulation study, or (2) a general algorithm presented without much information about its param-

eters. Thus Ross[1], in describing what is perhaps the most widely applicable and general approach to free-storage management, indicates the nature of the tools and techniques for very detailed programmer control of the management algorithm; the question of how to determine which algorithm to use is not considered. An excellent review and discussion of free-storage management is given by Knuth.[2] A recent paper by Campbell proposes a strategy based on the solution of the optimal-stopping problem on a Markov chain of fixed length, and shows that this strategy is superior to the first-fit method under certain conditions.[3]

The context in which the present free-storage management research was conducted is CP-67, a virtual machine control program, which provides for each logged-in user the environment of a System/360 computer.[4] Broadly speaking, our use of CP-67 can be viewed as a large general-purpose time-sharing system. It was in operation 24 hours a day, with 20 to 40 users logged-in during prime shift throughout the observation period. The user load is mainly interactive programming, e.g., editing, compiling, loading, and executing; however, batch computing does play an important part in the load on the system.

The general goal of the project was to explore techniques of system analysis, modeling, and redesign. The specific goal was not only to produce a free-storage management algorithm that would improve upon the existing system, but also to predict certain aspects of the improvement and then to validate and further quantify the improvement resulting from the new algorithm. The approach adopted was entirely empirical. The initial phase of the effort consisted of the monitoring, collecting, and reducing of trace data pertaining to the allocation and release of blocks of free storage. The second phase consisted of off-line development of, and experimentation with, various algorithms. The algorithm ultimately developed[5] depends heavily on the observed statistical properties of the data collected in phase one. Final validation and quantification of the improvement was effected through an on-line experiment with the two algorithms competing over an eight-day period. This methodological approach is general; free-storage management in the particular system is merely one of a class of problems to which it can be applied.

Background

free storage

CP-67 has three separate pools of free storage:

The page space: 4 K-byte blocks of the system's address space, made available via the system's relocation hardware on a dynamic basis.

SVC *linkage space:* 24-byte blocks of storage, used by the system for call linkages between its internal routines and modules.

Free storage: a pool of address space made up of separate, though possibly contiguous, 4 K-byte blocks. It is from this pool that the control program obtains storage for describing and controlling all system tasks. This pool can be extended (but not retracted) at the expense of page space.

storage management

Programs acquire storage blocks via the subroutine FREE (and return them via FRET) in integer multiples of double-words (8 bytes). Each such block is treated distinctly; if n double-words are acquired, n double-words are returned in whole, not in part. Further, each such block represents a specific request. Not only does a requirement for exactly this block size exist, but when the requirement has passed, the block is returned. Thus, there is neither splitting nor recirculating of the blocks by the requesting routine.

In its initial implementation, the FREE/FRET algorithm was very similar to one described by Knuth.[2] In brief, each block of free storage is threaded on a chain ordered by increasing addresses. A request is satisfied by the first exact fit found; if none is found, the first larger block is "split." On return, the block is rechained and "melded" with adjacent free blocks (if any). An unpleasant property of this procedure is that the depth of search as a result of a FREE (or FRET) is a random variable whose expectation and variance may increase to unacceptable levels during periods of heavy usage. Depending upon the load, this system managed a space of from 48 K to 100 K bytes during the period of observation, or from 7 to 14 percent of the total memory available to both free storage and page space. The range of requested sizes varied from 1 to 193 double-words; the time between request and release from milleseconds to hours. The FREE/FRET routine took between 2 and 45 percent of total control program (CP) time during 15-minute periods, with an average of 14.6 percent. (CP time is that time during which the CPU is in supervisor state.)

time versus storage space

The essence of the storage management problem is the desire to minimize simultaneously storage space wastage and control program overhead time, which are frequently diametric. One might hope to construct a crude relationship that would equate x wasted double-words with y units of wasted time (given the number of active users and the size of storage), but no work has been done on this.

Since it is unlikely that one can reduce both space and time inefficiencies simultaneously, and since no function equating space to time existed, the project goal set was to develop an algorithm to substantially reduce time inefficiency yet maintain roughly the same space requirements as the original algorithm.

Data gathering

motivation

The first step taken in the study was to monitor the real demands on the FREE/FRET routine. These data were crucial for two reasons. First, the data might possess striking statistical properties that would suggest a desirable algorithm; they did. Second, initial comparisons of any new algorithm with the original algorithm were to be made off-line to prevent system degradation. One standard approach to this type of comparison is to simulate the process. However, this suffers from the need to specify statistical models for the demands. Other researchers who have taken this approach have assumed relatively simple models for the demands, such as independence of successive requests, well-behaved distributions of request sizes and request durations, or independence of request size and request duration. It was important to determine if these simplifying assumptions were appropriate for our free storage problem; they were not.

An alternative to simulation is to capture free storage trace data from the running system. The sequence of varying requests and their subsequent releases can then be used as input to "drive" a program that duplicates the original algorithm or a specified variation of it. We use the term "emulation" to refer to this alternative to simulation.

During emulation, measurements were made to compare algorithms. The off-line emulation suggested that the algorithm ultimately developed would greatly improve upon the speed of the original algorithm. This was later validated by an on-line eight-day experiment.

Implementation

Since storage management is centralized in the FREE/FRET routine, it was a simple matter to trace each transaction made. A minor change was made in CP-67, which enabled the system operator to cause the capture on tape of the following data describing each call to FREE/FRET:

- Type of call (FREE or FRET)
- Time of day
- Location of call
- Size of block handled
- Location of block handled

In addition, each occurrence of an extension of free storage, i.e., the acquisition of 4K bytes from page space, was recorded. The data capture program used two 4096-byte buffers, and, in the interest of minimizing system slow-down, made no attempt to detect or prevent buffer overrun. Rather, the records were numbered serially, and during subsequent data reduction, a check was made for lost records.

Table 1 Block size request patterns

Order	*Size*	*Frequency*	*Cumulative frequency*	*Proportion*
1	4	121106	121106	0.248
2	5	106780	227886	0.467
3	29	75637	303523	0.623
4	1	54453	357976	0.734
5	8	54331	412307	0.846
6	10	19938	432245	0.887
7	3	18074	450319	0.924
8	9	10018	460337	0.944
9	18	9188	469525	0.963
10	17	4548	474073	0.972
11	7	2530	476603	0.978
12	23	1530	478133	0.981
13	6	1486	479619	0.984
14	2	1285	480904	0.986
15	12	979	481883	0.988
16	21	918	482801	0.990
17	11	657	483458	0.992
18	104	400	483858	0.993
19	146	386	484244	0.993
20	27	310	484554	0.994
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
71	77	1	487504	1.000
72	84	1	487505	1.000
73	98	1	487506	1.000
74	103	1	487507	1.000
75	109	1	487408	1.000
76	112	1	487509	1.000
77	124	1	487510	1.000
78	193	1	487511	1.000

System slow-down, although discernible, was within acceptable limits. A full reel of data (4800 buffers recording 960,000 calls to FREE/FRET) was typically obtained in about 2 hours, depending upon the level of activity. The only difficulty in acquiring a full trace of FREE/FRET calls was that those made during system start-up, before the trace program could be activated, were not recorded. It was assumed that after the first thirty minutes of emulated system operation, this loss was negligible.

data reduction

The raw data provided a great deal of relevant information for designing an efficient free-storage management algorithm. Essentially, the data describe a two-dimensional representation of the demands upon free storage in terms of storage space and storage time. The utility of such data derives in large measure from the marked demand patterns they depict and the strong reproducibility of these patterns from day to day.

size distribution

Perhaps the simplest summary of the data is provided by a tabulation of the number of requests for each block size, as shown in Table 1. These data were gathered on a particular day in the ap-

Table 2 Cumulative proportions of requests

Block sizes	*Days*			
	1	2	3	4
4	0.248	0.258	0.471	0.300
5	0.467	0.464	0.753	0.540
29	0.623	0.673	0.782	0.650
1	0.734	0.744	0.822	0.724
8	0.846	0.851	0.893	0.799
10	0.887	0.886	0.907	0.836
3	0.924	0.919	0.933	0.877
9	0.944	0.940	0.944	0.889
18	0.963	0.953	0.949	0.902
17	0.972	0.960	0.955	0.910
7	0.978	0.975	0.965	0.974
23	0.981	0.977	0.966	0.975
6	0.984	0.984	0.969	0.986
2	0.986	0.986	0.970	0.988
12	0.988	0.987	0.972	0.989

proximately 2.5-hour period from 9 a.m. to 11:30 a.m. The request sizes are ordered by decreasing frequency of requests, and reveal that a small number of sizes account for a large proportion of all requests. Thus, nearly 90 percent of all free storage requests in the 2.5-hour interval were accounted for by the half dozen sizes: 4, 5, 29, 1, 8, and 10. Addition of the next ten most frequently requested block sizes would bring this up to 99 percent. Table 2, which contains similar data for each of four different days, shows that there is a high degree of reproducibility. Although these data have been sampled from a single installation, the nature of the system tasks giving rise to frequently requested block sizes suggests that similar phenomena occur in other CP-67 installations. For example, blocks of sizes 4 and 5 are associated with user-initiated I/O, whereas blocks of size 29 are associated with paging I/O. Thus an efficient algorithm for CP-67 installations in general might be one that would service the above frequently requested sizes very efficiently in time and space without sacrificing reasonably efficient handling of the less frequently requested sizes.

demand duration

In addition to the relative frequencies of requests by block size, it is important also to know the storage time requirements of requests, i.e., how long various block sizes are used before they are returned. This information is presented in Table 3, which contains the frequency distributions of the durations that each block size is retained for data taken on day 4.

Most free-storage requests are short. For example, 94 percent of all blocks requested are returned in less than five seconds.

Table 3 Frequency of requests by size and duration

Size	Duration in seconds 0 < 5	5 < 10	10 < 30	30 < 60	60+	Proportion less than 5 seconds
65	0	0	1	0	21	0.000
17	1284	617	694	194	189	0.431
31	36	5	8	4	27	0.450
101	18	0	8	5	2	0.545
146	161	17	27	14	37	0.629
16	103	2	3	1	25	0.769
10	11761	1734	621	178	152	0.814
18	4395	862	23	0	0	0.832
3	13528	1672	488	83	135	0.850
4	106147	5510	4275	947	532	0.904
33	51	1	0	1	3	0.911
104	296	22	4	1	0	0.916
12	255	21	1	1	0	0.917
15	138	3	3	0	6	0.920
2	915	25	30	6	0	0.938
9	4543	212	31	3	0	0.949
14	209	10	0	0	0	0.954
5	90248	1678	1246	318	369	0.962
13	258	5	3	0	2	0.963
11	1032	39	0	0	0	0.964
6	4138	97	49	7	0	0.964
8	29028	176	3	1	0	0.994
1	28884	48	36	1	0	0.997
7	25065	63	2	1	0	0.997
29	43029	0	0	0	0	1.000
	367419	12821	7556	1767	1505	0.940

Note:
Any sizes not shown, and for which there are more than five observations, had all blocks returned in less than five seconds.

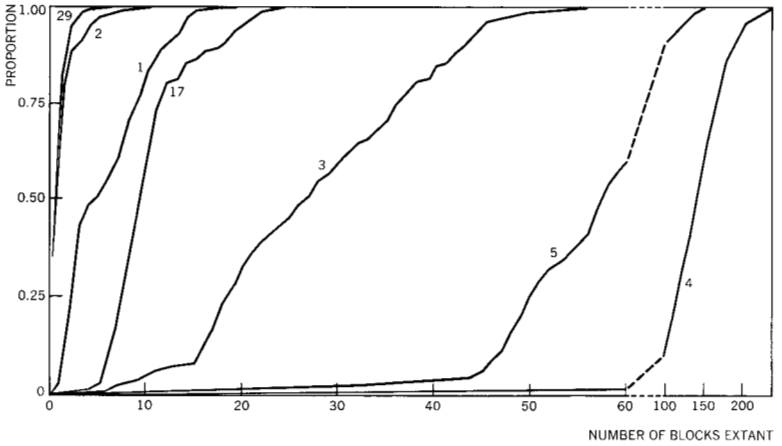
For many sizes, the requests are almost exclusively short. For example, sizes included in the ranges 1–2, 6–9, 11–15, 18–29, 32, 34–48, 50–63, 66–98, 103, and 109–129 are primarily short-term requests, as evidenced in Table 3. Only one block size gives rise to predominantly long occupancies—sixty five. The remaining block sizes, such as 3, 4, 5, 10, 16, 17, 31, and 146, which account for about two-thirds of all calls, are mixtures of long- and short-duration requests.

The interest in identifying long-duration block sizes derives from the potential profit in recognizing and servicing such requests in a separate area of free storage. This would reduce the fragmentation in main free storage and improve chances of returning empty pages to the control program during slack periods. This strategy was not pursued in the work reported here.

distribution of blocks extant

A third aspect of free-storage utilization is the distribution of the number of each block size extant. More explicitly, think of incrementing a counter whenever a particular size is requested and decrementing it whenever that size is returned. Then the distribu-

Figure 1 Cumulative distributions of numbers of blocks extant for selected block sizes



tion of the number of blocks extant for that size is simply obtained by a frequency count of the number of times the counter was equal to each of the integers 1, 2, · · ·. For example, Figure 1 shows the cumulative distributions of the numbers extant for sizes 1, 2, 3, 4, 5, 17, and 29. Such distributions are useful for estimating the probabilities that the number of blocks extant for given sizes will exceed some specified values. It is interesting to note that some sizes with relatively high frequencies tend to have only small numbers of requests extant at any time; and, conversely, some sizes with relatively low frequencies tend to have moderate numbers of requests extant at any time. Examples of these phenomena are sizes 29 and 17, respectively.

blocks extant by size and time

While data such as that presented in Figure 1 depict the variability of the number of blocks extant for individual sizes, they do not provide any information concerning the joint variation of the number of blocks extant for different sizes. Such information may be obtained by time-sampling the counters of the numbers extant for the various sizes. Table 4 shows a small portion of such data, recorded at ten-second intervals. Because of the transient nature of the system, in addition to information describing conditions at the end of the ten-second intervals, it is desirable to know something about activity during the intervals. One useful indication of such activity is provided by Table 5, which shows the maximum number of blocks extant for each of the 14 most frequently requested sizes during successive ten-second intervals. The data in Table 5 were generated from the same raw data used to generate the data in Table 4. The analysis of data such as in Table 5 was central to determining efficient free-storage management techniques.

Table 4 Number of blocks extant by size at 10-second intervals

	Time in ten-second intervals																			
Size	541	542	543	544	545	546	547	548	549	550	551	552	553	554	555	556	557	558	559	560
1	12	11	11	11	11	11	11	11	11	11	13	12	11	11	11	11	12	12	13	12
2	1	1	1	2	2	1	0	0	1	1	1	1	0	0	0	1	0	0	0	0
3	32	31	30	33	33	32	26	26	33	28	32	36	32	38	38	40	37	39	43	44
4	123	122	121	123	122	123	120	115	117	116	124	131	109	132	126	115	113	131	133	137
5	93	93	93	92	93	94	92	94	85	86	88	97	89	88	89	88	92	90	92	99
6	0	0	0	0	0	0	0	1	1	0	2	0	0	0	0	0	0	1	1	0
7	1	0	1	0	1	1	0	1	0	1	1	2	1	0	0	1	3	1	1	0
8	0	0	1	1	1	1	0	0	1	0	1	0	1	1	1	0	1	0	0	1
9	0	0	0	0	1	0	0	1	0	0	0	2	0	0	0	0	1	0	0	2
10	16	17	17	17	17	16	17	17	15	17	16	16	15	15	14	13	15	17	18	15
11	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
15	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
16	13	13	13	13	13	13	13	13	12	12	12	13	13	13	13	13	13	13	13	14
17	27	28	28	27	29	24	26	28	22	24	25	25	25	22	21	22	22	25	26	25
18	2	1	1	2	2	3	2	2	3	3	3	3	2	4	4	3	3	3	3	4
28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0	0	0	0	1	0	0	0	3	0	1	1	0	0
31	13	13	13	13	13	13	13	13	12	12	13	13	13	13	13	13	13	14	14	14
33	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
65	10	10	10	10	10	10	10	10	9	9	9	10	10	10	10	10	10	10	10	11
69	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
101	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
104	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0
129	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
146	5	7	6	6	6	7	5	5	6	4	3	5	5	5	6	6	6	6	7	6
TOTALS	359	357	356	360	364	359	344	346	337	333	355	375	335	362	358	345	351	373	386	393

Table 5 Maximum number of blocks extant during 10-second intervals for 14 most frequently requested sizes

	Time in ten-second intervals																			
Size	541	542	543	544	545	546	547	548	549	550	551	552	553	554	555	556	557	558	559	560
1	13	14	13	13	13	13	13	13	12	14	15	15	15	13	14	14	13	13	14	14
2	1	1	1	2	2	2	1	1	2	2	2	3	2	2	1	1	1	0	1	2
3	36	37	34	37	35	36	33	30	36	37	35	44	39	40	39	40	40	41	44	48
4	126	125	125	126	125	124	124	123	124	120	126	135	133	136	135	128	117	137	136	140
5	95	97	96	96	97	97	94	95	95	89	93	99	97	94	94	93	93	94	97	107
6	2	2	1	1	0	0	1	2	2	2	2	4	1	1	1	1	1	1	2	3
7	5	4	3	3	1	2	2	2	2	2	3	3	2	3	2	2	4	5	4	4
8	1	1	2	2	3	2	2	2	3	2	1	2	3	2	2	2	1	3	1	3
9	1	1	1	1	2	1	1	1	1	1	1	3	2	1	1	2	2	3	2	4
10	18	19	18	18	17	18	19	18	19	18	20	17	19	17	17	18	15	18	20	19
12	1	0	1	0	1	0	1	0	0	0	2	1	0	0	0	0	0	0	0	0
17	27	28	29	29	29	29	26	28	28	26	26	25	27	25	24	24	23	25	26	27
18	4	4	3	3	3	5	5	3	3	4	4	6	4	5	5	5	4	4	5	5
29	4	3	1	2	5	1	2	2	2	2	4	3	3	4	6	5	6	4	5	5

Emulation studies

The frequency distributions obtained during the data-gathering phase indicated that simplifying statistical assumptions, such as exponentially distributed duration times for the storage requests or independence of duration and size of the request, would not be tenable. This aspect of the problem made simulation an unattractive investigatory tool; there would be no simple way to characterize and then alter the demands on free storage and still be sure of maintaining a realistic study. Instead, as indicated previously, we chose to monitor the real demands upon free storage and use the recorded demands as input to a program that emulated and evaluated two broad classes of algorithms. (We continue to use the term emulation to refer to the use of real data from a running system as input to a program that duplicates a subset of the running system or specified variations of it.)

algorithm variations

The first class included the original FREE/FRET algorithm and variations of it, as follows:

- Specification of a constant to be used as a rounding-up factor in satisfying FREE calls, as opposed to exactly fitting the request (the constant equal to zero). This differs somewhat from rounding up storage requests to, say, the nearest multiple of a given quantum of storage, so as to reduce the number of different sizes requested. (We will comment on this latter idea later.)
- Elimination of melding or permission to meld across page boundaries on extended pages.
- Specification of the number of pages to be allocated to FREE storage at system start-up time.

pre-allocation

The second class of algorithms studied by emulation was one in which areas of free storage are pre-allocated for particular sizes. Sizes that are not pre-allocated are handled by a version of the original algorithm, as are pre-allocated sizes that overflow their assigned areas. The advantage of pre-allocation is that the handful of sizes that account for the vast majority of all free-storage requests can be handled quickly without a list search. One need only maintain a separate single-chained list of available blocks (a push-down stack) for each pre-allocated size. A request for a pre-allocated size is satisfied by the top element in the appropriate list. If none is available, it is serviced from the regular free-storage chain servicing nonallocated sizes. When a block from a pre-allocated area is no longer needed, it is returned to the top of the appropriate pre-allocation chain. Thus, by handling the top 5 or 10 sizes in separate chains, 90 to 95 percent of all requests are serviced in one step each.

One purpose of the emulator was to provide some indication of the trade-offs between time and space for these two classes of

algorithms. The space needed by a particular algorithm may be computed exactly for a given stream of input data. The translation from emulation time to real time, however, is quite complicated, since the emulator was written in FORTRAN, whereas any actual algorithm would be written in System/360 assembler language. Fortunately, the magnitude of the differences in list search lengths (depths) for the two classes of algorithms obviated the need for precise timing comparisons.

data reduction

After every block of 2000 transactions (FREE's plus FRET's), the emulation program provided the following data:

- Frequency distribution, mean, and standard deviation of depth of FREE search by size and for all sizes combined
- Frequency distribution, mean, and standard deviation of depth of FRET search by size and for all sizes combined
- Meld counts, number of unsuccessful melds, number of top only melds, number of bottom only melds, number of both top and bottom melds, and number of melds across page boundaries
- Frequency distribution of blocks in use, by size
- Number of double-words in use
- Frequency distribution of available blocks, by size, together with mean and standard deviation
- Number of available blocks
- Percentage utilization of available space

It should be noted that in computing the means and standard deviations of depth of search, both FREE's and FRET's of a size in a pre-allocated area were considered equivalent to a search of depth one. Thus, in terms of depth of search for FREE (or FRET), the optimally efficient algorithm would have an average search depth of one.

For algorithms of the second class, data taken on a typical day were used to provide specification of the pre-allocated sizes and of the number of double-words to be allocated for these sizes. Thus, when five sizes were to be pre-allocated, the top five most frequently requested sizes were determined from the frequency distribution by size (e.g., Table 1); then, the number of blocks to be allocated by size was determined as a specified percentage point, say the 50-percent point, of the cumulative distribution of the number extant by size. For example, in Figure 1, the 50-percent points of the distribution of block sizes 3 and 5 are 27 and 57, respectively. (This prespecification of sizes and percentages plus the entirely static nature of the algorithms of this class appeared to be the major drawback of the pre-allocation scheme.)

emulation results

The results of only one set of emulation experiments are reported. Five different algorithms were involved, one from the

Table 6 Average depths of FREE and FRET searches during blocks 3601–3800, and number of pages used after 3800 blocks

Algorithms	*Day* 1	2	3	4	5
	19.14	24.61	31.64	27.35	48.28
A	13.45	21.23	22.21	20.92	40.69
	12	12	15	12	17
	1.23	4.95	9.88	4.44	13.53
B	1.04	4.12	7.76	3.35	10.36
	12	12	15	13	17
	1.18	2.87	8.61	3.42	5.02
C	1.04	2.47	3.98	2.19	3.90
	12	13	16	15	18
	1.01	1.34	2.48	1.10	2.63
D	1.00	1.26	2.25	1.04	1.74
	12	14	17	15	18
	1.01	1.24	1.38	1.05	2.64
E	1.00	1.18	1.27	1.04	1.47
	12	14	16	15	18

Sizes allocated according to distribution of numbers of blocks extant observed on Day 1.

Algorithms
A–Control
B–50% points for top 5 sizes
C–95% points for top 5 sizes
D–95% points for top 15 sizes
E–95% points for top 15 sizes, with grouping

first class and four from the second. The algorithm chosen from the first class was the exact replica of the original algorithm of CP-67 and was labeled CONTROL. Algorithms 2 and 3 pre-allocated at the 50-percent and 95-percent levels, respectively, for the five most frequently requested sizes. Algorithm 4 pre-allocated at the 95-percent level for the top 15 sizes, whereas algorithm 5 partitioned these 15 sizes into subgroups by rounding up certain of the request sizes. These pre-allocated sizes and percentiles were determined from one day's tape of data, labeled Day 1. The five algorithms were then run against Day 1 and four other day's tapes, labeled Day 2 through Day 5. The responses measured were the average depth of search for the FREE and the FRET subroutines in consecutive intervals of 200 blocks, plus the number of pages used in the free-storage area. Table 6 depicts the results of the twenty-five emulation experiments. For purposes of comparison, the responses were recorded for blocks 3601–3800 on each tape, since the shortest available tape (terminated by a system malfunction) contained only 3800 blocks. Measurements at this point should be free of any transient start-up effects and should provide a sizeable sample for comparisons.

The results of the experiments indicate that increasing the extent of the pre-allocation for each size results in a marked reduction in depth of search for both FREE and FRET, at an initially small cost in the number of pages used in the free-storage area. For

example, results for algorithm 5 indicate that average search depths can be reduced by a factor of twenty-to-one at the cost of only 2 or 3 additional pages. These observations led to the design of a new algorithm, which has been implemented in CP-67 and is described later in this paper.

Dynamic chains and pooling studies

dynamic chain growth

The major drawback to the fixed allocation algorithms discussed is their dependence on static pre-allocation. Each time the system starts up, a chain of the exact same number of blocks is set up for each of the sizes to be pre-allocated. A realistic determination of the number of blocks needed in each chain requires a sizeable amount of data concerning the current state or degree of system usage. These algorithms cannot adjust dynamically to altered profiles of the installation's daily requests for storage nor, for that matter, to another installation's profile.

To eliminate the need for static pre-allocation in the algorithms of the second class, and thereby to make them more flexible, these algorithms were altered so that the chains servicing the specially treated request sizes would grow dynamically as the need for each particular size increased. The special sizes would never overflow their chains since the chains would increase in length without limitation to accommodate new requests. For example, sizes 4, 5, 8, 10, and 29 might be selected for special handling; these sizes would then be serviced with five separate growing chains. The remaining sizes would be treated in accordance with the original CP-67 algorithm.

This new hybrid class of algorithms, the third class, has the property that at any time, T, the chain for a specially treated size is as long as the maximum number of blocks of that size that have been extant up to time T since the most recent system start-up. This feature could ultimately cause a sizeable waste of storage space, which is the main trade-off in removing the static chain-length limitation. If this waste occurs after the days' peak of activity, it poses less of a problem, since time can be afforded for "garbage collection." The garbage collection procedures in the literature tend to be costly, since they require much time-consuming bookkeeping and checking of addresses. This aspect was not investigated; no specific garbage collection techniques other than that in the original CP-67 algorithm are discussed in this paper.

space allocation

The concept of allocation of storage space is easily quantified. Let $f_n(t)$ denote the number of blocks of size n extant at time t, let S represent the set of request sizes to be handled dynamically in their own separate chains, and let $t = 0$ be the time of the most

recent system start-up. The allocation of space at time T, $T > 0$, by the special chains is:

$$A_S(T) = \sum_{n \in S} n \cdot \max_{0 \le t \le T} f_n(t) \tag{1}$$

where $\max_{0 \le t \le T} f_n(t)$ is the largest number of blocks of size n extant from the last system start-up, $t = 0$, to the present time, $t = T$.

The waste of space at T due to the special chains is the allocation at T minus the need at T, namely,

$$\begin{aligned} W_S(T) &= \sum_{n \in S} n\{[\max_{0 \le t \le T} f_n(t)] - f_n(T)\} \\ &= \sum_{n \in S} n \cdot \max_{0 \le t \le T} [f_n(t) - f_n(T)] \end{aligned} \tag{2}$$

If one were to try to serve every size with its own chain, the waste of space would increase greatly. Much of this waste would be due to those sizes that rarely have a single block extant. If any of these sizes has appeared since the last start-up, then its chain has at least one block available, frequently unused.

The method chosen to combat the space waste caused by the special chains is to create nonoverlapping subpools, or disjoint subsets, among the request sizes contained in S and slated for special handling.

Each subpool is served by a single chain, where the blocks in the chain are as large as the largest request size in the subpool. Thus, if the requested sizes forming subpool p, $p = 1, 2, \cdots, s$, are denoted by $(n_{p1}, \cdots, n_{pk_p})$, with

$n_p = \max_r \cdot n_{pr}$, $r = 1, 2, \cdots, k_p$,

then the chain for the pth subpool consists of blocks of size n_p.

The allocation of space for the pth subpool at time T is:

$$A_p(T) = n_p \cdot \max_{0 \le t \le T} \sum_{r=1}^{k_p} f_{n_{pr}}(t) \tag{3}$$

The total allocation of space by the sizes in the pth subpool, when each is handled separately by its own chain, is:

$$\sum_{r=1}^{k_p} n_{pr} \cdot \max_{0 \le t \le T} f_{n_{pr}}(t) \tag{4}$$

No simple relationship such as an inequality exists between Equations 3 and 4, but a judicious choice of subpools should effect a reduction in space allocated, and hence, in space wasted.

The key question is how to determine the subpools so as to reduce the wastage reflected in Equation 3. Minimizing the wastage

for any one body of data is not practical because of the stochastic nature of the problem from day to day, and the desire to avoid unwarranted simplifying assumptions about the pertinent stochastic variables $[f_n(t)]$. Rather, the approach used was again empirical.

comparisons of subpool arrangements

One single tape of data was used to study the subpooling question with the intention not of finding a near-optimal subpooling arrangement, but rather of finding some simple rule of thumb that might generally be used to construct the subpools.

If there are no *a priori* restrictions on the number of subpools, or on any other aspect of the subpools, then the number of possible subpool arrangements is enormous. For a total of N sizes, the number of distinct subpool arrangements into M subpools can be shown to be:

$$\sum_{j=1}^{M} (-1)^{M-j} j^N \binom{M}{j} \Big/ M! = \sum_{j=1}^{M} (-1)^{M-j} j^N \frac{1}{j!(M-j)!} \tag{5}$$

many of which would be nonsensical in the problem at hand.

The total number of possible subpooling arrangements is Equation 5 summed over M, namely

$$\sum_{M=1}^{N} \sum_{j=1}^{M} (-1)^{M-j} j^N \frac{1}{j!(M-j)!} \tag{6}$$

Thus, for $N = 10$, the number of distinct subpool arrangements is 115,975. By the time $N = 25$, the number of arrangements is 4.64×10^{18}. Clearly, in the case under discussion, N is greater than 50 and, hence, an investigation of all subpools for even one day's tape is impossible.

One restriction that seems reasonable and decreases the number of distinct subpool arrangements considerably is to have subpools consist of consecutive sizes only. This is equivalent to the general procedure of rounding up request sizes. Again, for a total of N sizes, there are now 2^{N-1} distinct subpool arrangements that obey the requirement of "consecutiveness" or consistent "rounding up" of sizes.

The study of all 2^{N-1} arrangements is still an enormous undertaking, and so the empirical investigation was simplified further by checking possible subpooling arrangements among ten consecutive sizes at a time. Thus, for sizes 1, 2, 3, $\cdots$, 10, the $2^9 = 512$ possible consecutive subpools were constructed. Then the number of doublewords that would have been needed by each subpooling arrangement to accommodate the day's tape was computed. This procedure was repeated for sizes 10, 11, $\cdots$, 19, for sizes 19, 20, $\cdots$, 28, etc.

computation of space allocation

Some comments are in order on how the computation of the total usage of space was carried out for the 512 possibilities for each set of ten consecutive sizes. Each subpool arrangement of ten ordered sizes satisfying the requirement of consecutiveness can be represented by an ordered string of nine zeros and ones. The first number will be a one if the smallest and next smallest sizes in the set of ten sizes are to be pooled together, and a zero otherwise. The kth number will be a one if the kth and $k+1$st sizes in the set of ten are to be pooled together, and a zero otherwise. Thus, for example, one of the 512 subpool arrangements of the sizes $1, 2, \cdots, 10$ satisfying the requirements of consecutiveness is:

$$1, 2-3, 4, 5, 6-7-8, 9-10, \tag{7}$$

where sizes connected by a dash are to be pooled together. There are six subpools in this arrangement: sizes 1, 4, and 5 are not pooled with any other size; sizes 2 and 3 are pooled together; sizes 6, 7, and 8 are pooled together; and sizes 9 and 10 are pooled together. In terms of rounding up, size 2 is rounded up to 3; sizes 6 and 7 are rounded up to 8; and size 9 is rounded up to 10. The vector of zeros and ones representing this subpool arrangement of the ten sizes is:

$$0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1,$$

and the allocation of space up to time t is:

$$\begin{aligned}[1 \cdot \max_{0\le t\le T} f_1(t)] &+ \{3 \cdot \max_{0\le t\le T} [f_2(t) + f_3(t)]\} \\ &+ [4 \cdot \max_{0\le t\le T} f_4(t)] + [5 \cdot \max_{0\le t\le T} f_5(t)] \\ &+ \{8 \cdot \max_{0\le t\le T} [f_6(t) + f_7(t) + f_8(t)]\} \\ &+ \{10 \cdot \max_{0\le t\le T} [f_9(t) + f_{10}(t)]\}\end{aligned} \tag{8}$$

For the evaluation of space requirements, it is desirable to order the 512 arrangements to be checked so that in moving from the ith to the $i+1$st arrangement, there is either a single joining of two subpools at the ith stage into one subpool at the $i+1$st stage, or a single division of one subpool at the ith stage into two subpools at the $i+1$st stage. This minimizes the amount of computation needed to evaluate the allocation for each of the 512 arrangements. For example, if one had evaluated the allocation for the arrangement in Equation 7 and wanted the allocation for the arrangement:

$$1, 2-3, 4, 5, 6-7-8-9-10 \qquad (0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1),$$

namely,

$$\begin{aligned}[1 \cdot \max f_1(t)] &+ \{3 \cdot \max [f_2(t) + f_3(t)]\} + [4 \cdot \max f_4(t)] \\ &+ [5 \cdot \max f_5(t)] + \{10 \cdot \max [f_6(t) + f_7(t) \\ &+ f_8(t) + f_9(t) + f_{10}(t)]\}\end{aligned} \tag{9}$$

then one would have stored from the computation of Equation 8 all but the last term of Equation 9, which would require obtaining

$$\max_{0 \le t \le T} [f_6(t) + f_7(t) + f_8(t) + f_9(t) + f_{10}(t)]$$

One way to systematically run through all the possible 512 vectors of zeros and ones, i.e., through all subpool arrangements, so that at each stage either a single 0 is changed to a 1 or vice versa, is known in switching theory as the Gray code.[6] A program was written to follow the Gray code through the 512 possible arrangements, compute the storage space allocation for each, and then reorder the vectors of zeros and ones in terms of increasing space allocation. The output consisted of ten columns and 512 rows for each set of ten sizes studied, the first nine columns being the code for the specific subpool arrangement evaluated and the tenth column containing the corresponding storage allocation. Thus, the first row had the smallest requirements, the second row the next smallest, and so on. Table 7 contains a portion of the output for sizes 10, · · ·, 19. Looking at this output, if the jth column contained a relatively large number of zeros toward the top of the output, this suggested that the jth and $j + 1$st sizes should not be pooled together; similarly a large number of ones toward the top of the jth column of the output suggested that the jth and $j + 1$st sizes should be pooled together. Based on this computation and subsequent analyses, a simple rule of thumb for subpool construction was postulated.

rounding up of request sizes

The general idea of rounding up the size of a storage request is not new. It has, for example, been studied via simulation by Randell.[7] He investigated the special case of rounding up request sizes to the nearest multiple of a given quantum of space, e.g., to the nearest power of 2. His conclusion was that rounding up in general is not desirable in that it offers no gain in storage utilization.

The proposed rule for subpool construction by rounding up requests requires sampled data on the number extant for each size over a reasonable and representative period of time, data similar to that in Table 5. Each size that typically has a large number extant will be called a mode of the distribution of the number extant. The proposed rule of rounding then is to round a given size up to the nearest mode. This is called "modal roundup." In Table 5, the modes are sizes 4, 5, 3, 17, 10, and 1 in decreasing order of the number extant.

This rule contains a negative statement in that rounding up modes is to be avoided. Intuitively this seems reasonable; if a size with a large number of requests typically extant is rounded up, wastage will, in all likelihood, increase.

Table 7 Storage utilization for different subpool arrangements

	Sizes									Usage
	11	12	13	14	15	16	17	18	19	
	0	0	0	1	1	1	1	1	0	1389
	0	1	1	0	1	1	1	1	0	1389
	1	0	0	1	1	1	1	1	0	1389
	0	0	0	1	1	0	0	1	0	1391
	0	1	1	1	1	1	1	1	0	1391
	0	1	0	1	1	1	1	1	0	1391
	1	0	0	1	1	0	0	1	0	1391
A	0	1	1	1	1	1	0	1	0	1393
R	0	1	0	1	1	0	0	1	0	1393
R	0	0	1	1	1	0	0	1	0	1394
A	1	0	1	1	1	0	0	1	0	1394
N	0	0	0	1	1	0	1	1	0	1395
G	0	0	0	1	1	1	0	1	0	1395
E	0	0	1	1	1	1	1	1	0	1395
M	1	0	1	1	1	1	1	1	0	1395
E	1	0	0	1	1	1	0	1	0	1395
N	1	0	0	1	1	0	1	1	0	1395
T	0	0	0	0	1	1	1	1	0	1397
S	0	1	0	1	1	1	0	1	0	1397
	0	1	0	1	1	0	1	1	0	1397
	1	0	0	0	1	1	1	1	0	1397
	0	0	1	1	1	0	1	1	0	1398
	0	0	1	0	1	1	1	1	0	1398
	1	0	1	0	1	1	1	1	0	1398
	1	0	1	1	1	0	1	1	0	1398
	0	0	0	1	0	1	1	1	0	1399
	0	0	1	1	1	1	0	1	0	1399
	0	1	0	0	1	1	1	1	0	1399
	1	0	1	1	1	1	0	1	0	1399
	1	0	0	1	0	1	1	1	0	1399
	0	0	1	1	0	1	1	1	0	1401
	0	1	0	1	0	1	1	1	0	1401
	1	0	1	1	0	1	1	1	0	1401
	0	1	1	1	1	0	0	1	0	1402
	0	0	0	1	0	0	0	1	0	1404
	1	0	0	1	0	0	0	1	0	1404
	0	0	0	1	0	1	1	0	1	1405
	1	0	0	1	0	1	1	0	1	1406
	0	0	1	1	0	0	0	1	0	1406
	0	1	1	1	1	0	1	1	0	1406
	0	1	0	1	0	0	0	1	0	1406
	1	0	1	1	0	0	0	1	0	1406
	0	0	0	1	0	1	0	1	0	1407
										.
										.
										.
	0	0	0	0	0	0	0	0	0	1483 (no pooling)

If one holds to the modal round-up rule, then one can see the unattractiveness of rounding up to the nearest multiple of a given quantum Q as reported by Randell. One will almost surely round up many modes in the process, thereby increasing the storage waste.

Beyond what is implied by modal round-up, there appears to be no generality that can be stated about rounding up request sizes

except, perhaps, that there appears to be some insensitivity of storage usage to the rounding up of many of the rarely occurring smaller sizes.

Finally, it should be noted that the gains in space allocation by good subpooling over no subpooling are nowhere as dramatic as the gains in time by special chaining of certain sizes over no special treatment of any sizes.

On-line experimentation

the experiment

The next step in this investigation was the ultimate validation of the emulation findings concerning dynamic chains and subpooling for the most frequently observed sizes. This was an on-line experiment comparing the original free-storage algorithm's performance with the performance of an algorithm of the third class, with dynamically growing chains for certain subpooled sizes.

The specific new algorithm chosen for the experiment provided special service for the 13 sizes most frequently used in our system. These sizes were 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 17, 18, and 29; they accounted for approximately 98 percent of all the calls to free storage. The subpooling employed for the thirteen sizes was: 1, 2–3, 4, 5, 6–7–8, 9–10, 17–18, 29. This arrangement satisfies the modal round-up rule by not rounding up the modes in Table 4, with the exception that 17's were rounded up to 18. The desirability of this exception is readily apparent from Table 7, which clearly shows that 17's and 18's are grouped together in most of the better arrangements.

Since those sizes not assigned to subpools by the new algorithm are handled according to the rules governing the original algorithm, the latter algorithm can be obtained as a special case by not specifying any sizes for subpool handling. The ease of changing algorithms by parameter settings thus facilitates comparative evaluation via designed experiments carried out in an operational environment. An experiment of this nature was conducted shortly after incorporation of the new algorithm. At 5:00 a.m. each morning during a two-week period, a CP-67 system containing either the new or old FREE-FRET algorithm was loaded according to a specified schedule and allowed to run for 24 hours. If a system abnormal termination occurred during the day, the same system would be reloaded. The experiment was designed to consist only of eight days, namely Monday through Thursday of each week. The reason for this was twofold. First, weekend data was excluded due to low system usage during such periods. Furthermore, in order to have a statistically balanced experiment with respect to sequencing of algorithms within a week, and day by day comparisons between weeks, an even number of days in

Table 8 Experimental design

	Monday	*Tuesday*	*Wednesday*	*Thursday*
Week 1	old	new	new	old
Week 2	new	old	old	new

Table 9 Experimental results

	CP time per call (microseconds)			
	Monday	*Tuesday*	*Wednesday*	*Thursday*
Week 1				
FREE	325	*50*	*45*	426
FRET	221	*38*	*34*	287
Week 2				
FREE	*37*	346	313	*45*
FRET	*27*	236	217	*33*

Note: Italicized numbers correspond to the new algorithm.

each week was required. The design of the experiment is shown in Table 8.

This design provides protection against possible linear or quadratic time trends in usage during the course of the experiment, and allows comparisons between algorithms, among days of the week, and between weeks. During the experimental period, data on forty different variables were recorded at five-minute intervals by means of a software monitoring program. These included cumulative counts of FREE and FRET calls broken down by subpool class, free-storage occupancy statistics, and accumulated supervisor time spent in FREE and FRET.

These measurements were provided specifically to compare the two FREE/FRET algorithms. Additionally, data normally collected on the system for measurement and analysis of system performance were recorded during the experimental period. These data included counts of various types of paging and I/O activity, numbers of active and inactive users, and subdivisions of CPU time into several different states.

experimental results

The principal results of the experiment are summarized in Table 9, which shows the average supervisor time per FREE and per FRET call during each day of the experiment. The data are presented according to the layout of Table 8, with results corresponding to the new algorithm italicized for ease of comparison. It is readily apparent that the average CP time per call to FREE or FRET is reduced by a factor of 7 or 8 to 1 by the new algorithm.

Table 10 Estimated CP time per I/O (milliseconds)*

	VSIO	*Page I/O*	*Spool I/O*
Old	9.7	3.0	5.7
New	7.9	2.0	4.6

Abstracted from Table 11, Reference 8

Table 11 Average throughputs during test period*

	Old	*New*
Percentage of problem state time	19.8	22.0
Percentage of CP state time	21.3	18.1
VSIO per sec	11.7	12.4
VMIO per sec	8.5	8.5
SPOOL I/O per sec	1.1	1.2
PAGE I/O per sec	6.7	7.0

*Taken from Table 12, Reference 8

These differences are so large that formal statistical analyses and tests of significance are scarcely required to verify the conclusions.

Finally, the question of how the new FREE-FRET algorithm affects overall system performance naturally arises. A simple answer to this question, provided by direct system measurement during the experiment, is that CP time spent in FREE/FRET has been reduced from an average of 14.6 percent of total CP time to an average of 2.4 percent. A more extensive answer to the question may be gleaned from Tables 10 and 11, which have been abstracted from a paper by Bard.[8] Table 10 shows the reductions in CP time for various types of I/O operations, as estimated by fitted regression models, while Table 11 shows how throughput rates have been increased by the introduction of the new algorithm. As a result of the study, the improved algorithm was incorporated into CP-67. The salient feature of the study was the demonstration that significant improvement in system performance can be effected by designing resource allocation algorithms to take advantage of observed demand and utilization patterns.

ACKNOWLEDGEMENT

The authors gratefully acknowledge R. Adair, J. Ravin, and J. Seymour for their suggestions and assistance in this research. In particular, J. Ravin wrote the data reduction and emulation programs and carries the major burden of the detailed validation of the emulation and the sample data used to drive it, and J. Seymour designed and programmed the FREE-FRET algorithm in-

corporated in CP, together with the software measurements required in the on-line experiment.

REFERENCES

1. D. T. Ross, "The AED free storage package," *Communications of the ACM* **10,** No. 8, 181–192 (August 1967).
2. D. E. Knuth, "Dynamic storage allocation," *The Art of Computer Programming* **1,** Addison-Wesley, Reading, Massachusetts (1968).
3. J. A. Campbell, "A note on an optimal-fit method for dynamic allocation of storage," *The Computer Journal* **14,** No. 1, 7–9 (February 1971).
4. R. A. Meyer and L. H. Seawright, "A virtual machine time-sharing system," *IBM Systems Journal* **9,** No. 3, 199–218 (1970).
5. This algorithm was incorporated into CP-67, version 3.
6. W. S. Humphrey, Jr. *Switching Circuits with Computer Applications,* McGraw Hill, New York, 109 (1958).
7. B. Randell, "A note on storage fragmentation and program segmentation," *Communications of the ACM* **12,** No. 7, 365–372 (July 1969).
8. Y. Bard, "Performance criteria and measurement for a time-sharing system," *IBM Systems Journal* **10,** No. 3, 193–216 (1971).