Analysis of free-storage algorithms—revisited

by G. Bozman
W. Buco
T. P. Daly
W. H. Tetzlaff

Most research in free-storage management has centered around strategies that search a linked list and strategies that partition storage into predetermined sizes. Such algorithms are analyzed in terms of CPU efficiency and storage efficiency. The subject of this study is the free-storage management in the Virtual Machine/System Product (VM/SP) system control program. As a part of this study, simulations were done of established, and proposed, dynamic storage algorithms for the VM/SP operating system. Empirical evidence is given that simplifying statistical assumptions about the distribution of interarrival times and holding times has high predictive ability. Algorithms such as first-fit, modified first-fit, and best-fit are found to be CPU-inefficient. Buddy systems are found to be very fast but suffer from a high degree of internal fragmentation. A form of extended subpooling is shown to be as fast as buddy systems with improved storage efficiency. This algorithm was implemented for VM/SP, and then measured. Results for this algorithm are given for several production VM/SP systems.

An efficient, dynamic storage allocation algorithm is essential to the performance of complex software systems. These systems require the ability to reuse areas of memory for such things as control blocks, buffers, data areas, and state vectors. The reuse ability is needed in order to keep the total memory requirement reasonable. Without reuse it would be necessary to permanently assign enough storage for each purpose to ensure a very low probability of exhausting each storage type. Because the frequency of obtaining storage may happen more than one thousand times per second, throughput may be affected by the processing time required. Unfortunately, storage efficiency and CPU efficiency are usually tradeoffs in the selection of an algorithm.

Storage inefficiency is a result of fragmentation, both internal, which is the result of giving out more storage than requested (e.g., by rounding up to some boundary), and external, which is the "checkerboard" effect caused by alternating blocks of available and in-use storage. CPU inefficiency results when it becomes necessary to search for a block that will satisfy a request for free storage (or the proper place to "insert" a released item), and can be measured by the mean number of blocks inspected per request (release).

Early work in this area focused on the relative efficiency of various strategies that process requests against a linked list of available storage blocks. The algorithm known as "first-fit" consists of searching the available list and accepting the first free area that is greater than or equal to the required size. When a suitable block is found, it is split into a block of the right size that will be used and a fragment that is left on the free list. The "best-fit" strategy consists of searching the entire free list in order to find a free block that if split will leave the smallest fragment. Naturally, if an exact fit is found, the search is terminated. There are two variations of best-fit that are distinguished by whether the first or last of equal best-fitting blocks is used. In "worst-fit" the free block that results in the largest fragment is chosen, except that an exact fit is taken when found.

Collins[1] simulates first-fit, best-fit, worst-fit, and random-fit, finding first-fit slightly better than best-fit. Iliffe and Jodeit[2] describe the use of codeword descriptors to provide mobility of data in a first-fit environment with garbage collection.

Knowlton[3,4] and Markowitz et al.[5] independently developed the binary buddy system. Buddy systems maintain space in separate pools by size (e.g., powers of two in the binary buddy system). Requests are rounded up to the appropriate size boundary, resulting in internal fragmentation. If a block of that size is not available, an iterative search is made of successively larger-sized pools until a block is found. This block is then iteratively split into "buddies" until a block exists for the requested size. When blocks are released, buddies are recombined if possible.

Buddy systems maintain space in separate pools by size.

Ross[6] describes the Massachusetts Institute of Technology AED free-storage package that uses zone partitioning as a solution for external fragmentation. Each zone independently manages its inventory. Randell[7] demonstrated via a simulation study that, if requests are rounded up in an attempt to reduce the memory loss due to external fragmentation, the internal fragmentation loss rapidly predominates. Knuth[8] gives an excellent review of previous work, provides simulation results supporting first-fit over best-fit, describes a modified first-fit algorithm that starts each search with the block after the last one given out (thereby cycling through the linked list of available blocks), and recommends the investigation of the Fibonacci buddy system. Campbell[9] describes an optimal-fit algorithm, based on the optimal-stopping problem on a fixed-length Markov chain, which attempts to combine the best property of first-fit (speed) with the best property of best-fit (reduced external fragmentation).

Purdom, Stigler, and Cheam[10] simulate first-fit, binary buddy, and segregated storage (variable-sized subpools with splitting and recombination—similar to the generalized Fibonacci buddy). The binary buddy is found to be the fastest. Robson[11] proves that for any nonrelocating strategy the amount of storage required is bounded below by a function that rises logarithmically with the size of the blocks that are used. Margolin, Parmelee, and Schatzoff[12] describe a study that led to an improved algorithm for the computer control system CP/67. Since their work strongly influenced the algorithm currently used on the Virtual Machine/System Product (VM/SP) and since our work can in some ways be considered an extension of theirs, their work is a key antecedent to this study.

Hirschberg[13] follows Knuth's suggestion and does a simulation study of the Fibonacci buddy system *vis-à-vis* the binary buddy system, concluding that the Fibonacci is superior. Fenton and Payne[14] simulate first-fit, Knuth's modified first-fit, best-fit, half-fit, and worst-fit. Their study concludes that best-fit is superior; first-fit and half-fit are somewhat better than modified first-fit; and worst-fit is worst. Shen and Peterson[15] develop a weighted buddy system that provides more sizes than the binary buddy. They find that internal fragmentation is decreased often at the expense of some increase in external fragmentation and conclude that the weighted buddy system will give good results if the request distribution is primarily composed of small sizes.

Russell[16] gives mean value bounds for the overallocation due to internal fragmentation in a one-level buddy system. Shore[17] finds through simulation that first-fit and best-fit are generally within one to three percent of each other in terms of memory utilization. He provides strong evidence that the relative performance of the two strategies depends on the frequency of requests that are large compared with the average request. In terms of the coefficient of variation, α, of the request distribution, he finds that first-fit outperforms best-fit when α is greater than or equal to one. Bays[18] confirms Shore's results and also finds modified first-fit to be inferior when the mean request size is less than one-sixteenth the total available memory. Cranston and Thomas[19] describe a simple recombination scheme for the Fibonacci buddy systems. Ferguson[20] defines a generalized Fibonacci scheme and provides tables that are useful in the generation of these systems. Peterson and Norman[21] study the binary, Fibonacci, and weighted buddy systems and derive the internal fragmentation for each for the uniform request distribution. They also provide simulation results that suggest that,

although the internal fragmentation varies, the total fragmentation (i.e., internal and external) is about the same for these three buddy systems.

In this paper a simulation study is discussed that compares many dynamic storage allocation strategies in medium and large time-sharing environments. This work resulted in an algorithm that significantly improves the performance of large Virtual Machine/System Product Conversational Monitor System (VM/SP CMS)[22] [VM1, CMS1] systems.

This paper first discusses the environment in which this work was done. Next, the simulation methodology is explained. Then simulation results and performance results for two systems are presented. Finally, conclusions are drawn from this study.

Background

The impetus for this study was provided by hardware and software monitor data collected at the Data Centers of the IBM Thomas J. Watson Research Laboratory and the former IBM Office Products Division (OPD) headquarters. These data indicated that the VM/SP dynamic storage algorithm consumed 11 to 20 percent of the supervisor-state CPU on the OPD 3033 Uni-Processor (UP) and the Research 3033 Multi-Processor (MP). Under peak load the percentage was at the high end of this range. The high CPU time caused high lock holding time for the primitive lock on the MP that prevents the concurrent execution of the dynamic storage allocator. The high lock holding time in turn caused longer lock wait time for the other processor. The wait time on this lock was one to one and a half percent of elapsed time during typical load.

The instruction that referenced the next address in the linked list of available storage blocks was most frequently seen in the hardware monitor samples. This instruction frequently "missed" cache because of the relatively large area of memory containing the linked list. It appears that searching linked lists tends to subvert the cache by (1) yielding a low hit-ratio during the search and (2) leaving the cache full of data that are very unlikely to be referenced after the search.

This study was performed by independent groups working at each data center. The OPD group was working on a simulation study, whereas the Research group was studying the effects of modifications to the existing VM/SP algorithm. When the two groups became aware of their common interest, their shared discoveries led the way to the final solution.

The fact that there were two independent groups using different tools and techniques has resulted in some inconsistencies in the analysis that was done on the systems that were studied. However, all applicable results are provided. If a specific result is not provided for a given system, it is because the analysis was not performed.

In order to understand the work reported in this paper, it is necessary to understand the dynamic storage allocation algorithm that was used in VM/370 at the time of this work. All other algorithms that were studied were measured against this. It proceeds as follows.

Ten stacks, each three doublewords wide (one doubleword is eight bytes of eight bits each), are maintained for free storage elements less than or equal to 30 doublewords such that the first stack services requests from one to three doublewords; the second, four to six doublewords; to the tenth, 28 to 30 doublewords. All requests within this range are rounded up to the appropriate boundary. These stacks were called subpools by Margolin et al., and we will use that term here. Initially all of the subpools are empty. The purpose of the subpool is to be able to find a free storage element immediately, thus eliminating any searching of a linked list at all.

If the request cannot be satisfied by a subpool, either because the subpool is empty or the request is greater than 30 doublewords, a search is made of a linked list of available storage blocks. This list is called the free list and is maintained in order of increasing address. The list is ordered by storage address in order to facilitate the coalescing of a newly freed block with adjacent free blocks. Initially the free list consists of one block which represents the storage dedicated as dynamic storage at system generation (an integral number of 4096-byte pages). If the free list is empty or cannot satisfy the request, a page (or multiple pages if the request exceeds 4096 bytes) is "borrowed" from the page pool that provides real memory to users and the operating system itself. However, if the request that cannot be satisfied is less than or equal to 27 doublewords, an attempt is first made to "split" a larger subpool block into smaller sizes. If the split is not possible, a page is borrowed from the page pool with the intention of returning it as soon as possible. The act of borrowing

is called "extending," and the borrowed page is called an "extended" page. Whenever a release of a block causes an extended page to be completely contained in the free list, it is returned to the page pool. Because the dedicated dynamic storage is initially generated at the top of memory (i.e., highest addresses), all available blocks from extended pages will be at the front of the free list.

The searching of the free list proceeds as follows:

- If an exact fit is found with a nonextended block, the search is terminated and the block is used to satisfy the request.
- If an exact fit is not found and the request size is less than or equal to 30 doublewords, the low end of the first larger nonextended block is used. If there is no nonextended block greater than or equal to the requested size, the last equal or larger extended block is used with an equal size taking precedence. If it is necessary to use a larger block, the low end of this block is given out.
- For requests greater than 30 doublewords a similar strategy is used, except that the high end of the last larger block is used in the case where there is no exact match.

Release of storage to the free list proceeds as follows:

- Upon release, blocks less than or equal to 30 doublewords are pushed into the appropriate subpool.
- Items greater than 30 doublewords are inserted directly into the free list. Proper placement in the free list requires a serial search through the address-ordered list until the correct place is found. If they are adjacent to another free block(s), the blocks are coalesced into a single larger block.

In the periodic emptying of the subpools, whenever any user leaves the system, or at least once an hour, all blocks in the ten subpools are removed and inserted into the free list. This procedure allows for downward adjustment of the number of blocks after an unusual requirement for a particular size, and it allows extended pages to be returned.

The ten-subpool strategy is derived from Margolin et al., who found that over 99 percent of all requests in CP/67 were in this size range. This study found that this percentage has deteriorated with time (e.g., from 93.4 to 97.0 percent on the systems that we studied) and is currently a function of the release level of VM/SP, local modifications, and local request distributions.

This linked-list allocation strategy segregates "small" requests at the low end of the dynamic storage block and "larger" requests at the high end, except that an exact match is always taken regardless of where it is in storage. This is a variation of best-fit that attempts to control external fragmentation even further. Because it is an interesting strategy independent of the subpools, it is included in the simulation study to see how it compared with the traditional first-fit and best-fit methods.

Method

It was decided that the only reasonable way to study the effect of various dynamic storage algorithms on VM/SP was a simulation study. The alternative of multiple changes to the real system was rejected as being inflexible and risking severe performance degradation and system outages.

Next it was necessary to decide how to represent the VM/SP dynamic storage environment to a simulator. Margolin et al. rejected the use of simplifying statistical assumptions in the request and holding distributions and elected instead to modify CP/67 to log requests to tape. In this way they could rather accurately (some requests were lost because of buffer overruns) recreate the dynamic storage environment for a given day. We were concerned that this approach would not be feasible in our environment primarily because the high request plus return rates (over 2000 per second) would force a difficult tradeoff between significant data loss and perturbation of the system. In addition, we wanted to develop a technique that would be more flexible in that it would not require large amounts of data as input.

The successful use of simplifying assumptions, such as exponentially distributed service times, in queuing network analysis (see Buzen[23]) encouraged us to use the following method:

- VM/SP was modified to collect, for each size requested, the mean number of blocks outstanding (i.e., in use) and the total number of requests at any point in time. The mean number of blocks in use was computed by incrementing a vector element (corresponding to the size) for each request satisfied and decrementing for each return. At any instant in time this element is equal to the number of blocks outstanding for the size. The total number of requests was computed by incrementing a vector element for each request.

- A sampling program was written to compute from these data the mean interarrival time in seconds, mean holding time in seconds, and mean number of blocks outstanding for each size over any time interval. This was done as follows:

  1. Mean interarrival time (in seconds), $t = s/(r_1 - r_0)$
     where
     s = sample interval in seconds
     r_0 = total number of requests from system startup (i.e., IPL) to the start of the sample interval
     r_1 = total number of requests from system startup (i.e., IPL) to the end of the sample interval

  2. Mean holding time (in seconds) = Nt
     where
     N = mean number of blocks outstanding (computed from samples of the vector element)

     The relationship of the mean interarrival time, the mean holding time, and the mean number of blocks outstanding is similar to that defined by Little's result in queuing theory,[24] which gives the relationship between the mean number of customers in a queuing system (L), the mean arrival rate (λ), and the mean time (W) spent in the system as $L = \lambda W$. Consider a customer picking up a cart at the entrance of a supermarket and using the cart as he makes his way through a number of servers, finally returning it at the entrance as he leaves. We can make an analogy between the supermarket customer and a time-sharing user. The cart then becomes the dynamic storage required to support the user's sojourn through the time-sharing system (i.e., queuing system), and the average arrival rate and average residency time of dynamic storage in the system will have a direct correlation with those statistics for users. A similar analogy can be made on a micro scale for the dynamic storage used for events such as I/O operations.

Having the mean interarrival time and mean holding time for each size, we then made the simplifying assumption, in the simulation study, that the interarrival times and hold times were exponentially distributed. There are allocation phenomena—such as the "simultaneous" creation (deletion) of different-sized control blocks when a user logs on (logs off) the system—that are not properly modeled by a stochastic process. However, similar phenomena occur in those aspects of computing systems that are modeled with reasonable accuracy by queuing networks, and we were hopeful that we would achieve results of similar accuracy. The later validation of the simulation results, by comparison to measurements of real systems, supports the use of these assumptions (see section on system results).

In order to provide the data structures necessary to simulate the various algorithms of interest, a discrete event simulator was written in Pascal. This program performs the following functions:

1. Reads (from a parameter file) the size of dedicated dynamic storage, mean interlog-off time (used by the standard VM/SP and subpooling algorithms), and, for each size, the mean interarrival time and the mean holding time.
2. Initializes the simulation by scheduling the stop event, first sample event, first checkpoint event, and, for each size, the first request. Also, if applicable, the first user log-off event was scheduled. A different pseudo random-number generator was used for the log-off event so that all algorithms would have the same series of storage requests and releases.
3. Maintains the event list using a time-indexed method.[25]
4. Provides checkpoint and restart capability.
5. Provides a sampling and statistics generation facility.
6. Provides a uniform interface to external routines to handle storage request, storage release, and user log-off events.

Each algorithm was written as a separate subprogram that was called by this main program to service an event such as a dynamic storage request, dynamic storage return, or user log-off (if appropriate to the algorithm).

Results

Simulation results. The following VM/SP systems were modified to collect parameters for the simulation study:

- FRKVM1, a 3033 UP serving an average of 280–340 logged users at the former Office Products Division Headquarters in Franklin Lakes, New Jersey.
- YKTVMV, a 3033 MP serving an average of 450–540 logged users at the Thomas J. Watson Research Center in Yorktown Heights, New York.

- CAMBRIDG, a 158 UP serving an average of 40–50 logged users at the IBM Cambridge Scientific Center in Cambridge, Massachusetts.

The three tables in the Appendix give the dynamic storage parameters for each of these systems. The mean interarrival and holding times are given in seconds for each doubleword size. Sizes not listed did not have any activity during the parameter collection period. Ten milliseconds was the minimum holding time used for the simulation. The minimum holding time was introduced because there were some storage sizes that had infrequent requests with short durations. The accuracy of the sampled holding time for these sizes was poor, and it sometimes led to unreasonably short holding times. These requests were so infrequent that the adjustments are not a factor in the results. Requests greater than 512 doublewords were rare (less than 0.007 percent of all requests at FRKVM1 and less than 0.017 percent at YKTVMV and CAMBRIDG) and have been combined with the 512-request data.

At Franklin Lakes, samples were taken during a two-hour period of typical afternoon load. The free-storage vectors were sampled at 30-second intervals. In order to see if this sampled distribution would yield simulation results that were analogous to those measured on the real system, three of the simulation metrics were compared with samples from days of FRKVM1 activity that had a user load similar to that during which the parameters were collected. The simulation results were found to be typical of those measured on FRKVM1. The three days given in Table 1 are representative and illustrate the range of the measured data.

At Yorktown Heights and Cambridge, samples were collected during similar periods of typical load, but were not correlated to the activity on separate days as at Franklin Lakes. However, the storage utilization and mean free list size were within the ranges witnessed on the real systems.

Tables 2–4 give the simulation results for each system. The metrics used in this study are defined as follows:

- Mean items visited per request: The minimum possible value is 1.0. This is the primary measurement of CPU efficiency. In a linked-list strategy, this is the number of items visited on the list. An item popped from a subpool that is singly linked is counted as one visit. An item popped from a subpool that is doubly linked (e.g., buddy systems) is counted as two visits if the subpool is not left empty by its removal, otherwise as one visit.
- Mean items visited per release: The minimum possible value is 1.0. In a linked-list strategy, this is the number of items visited on the list in order to find the proper place to insert the released item. An item pushed into a singly linked subpool is counted as one visit. An item pushed into a non-empty doubly linked subpool is counted as two visits.
- Subpool hit ratio: The ratio of requests that were satisfied by a subpool block to the total number of requests. This is only applicable to algorithms that use some form of subpooling.
- Mean free-list size: The mean number of items on the linked list of available storage blocks. This size is an indicator of external fragmentation and, in many algorithms, directly affects the mean number of items visited.
- Extend rate: The mean number of requests per minute for an extended page that occurred during the last hour of simulated time. Higher rates incur higher CPU overhead.
- Extended pages: The mean and maximum number of extended pages that were required above the initial dynamic storage allocation. Note that there is no relationship between the extend rate and the number of extended pages. For example, the repeated request and release of one extended page will result in a high rate but low (less than one) mean number of extended pages. Some algorithms hold extended pages longer than others and consequently have a lower extend rate.
- Mean storage out: The mean number of pages in use (given out but not yet returned) during the simulation.
- Storage efficiency: The ratio of the mean requested (i.e., before any rounding up) storage in use to the mean storage required by the algorithm. That is,

Table 1 FRKVM1 simulation versus observed values

Source	Mean Subpool Hit Ratio	Mean Freelist Length	Mean Pages Used*
sample day 1	0.938	569	613
sample day 2	0.946	695	584
sample day 3	0.933	884	672
simulation	0.943	704	630

*mean pages used = dedicated dynamic storage pages + mean extended pages.

Table 2 FRKVM1 simulation results (3033 UP)

Algorithm	Mean Items Visited per Request	Mean Items Visited per Release	Subpool Hit Ratio	Mean Freelist Length	Extend Rate (pages/min)	Extended Pages Mean-max	Mean Storage Out (pages)	Storage Efficiency
standard	30.6	17.8	0.943	704	287	130–159	591.2	0.896
first-fit	949.2	896.8		3157	309	117–145	564.2	0.914
best-fit-last	245.0	186.0		1108	266	94–126	564.2	0.947
best-fit-first	273.5	204.0		1162	264	94–127	564.2	0.949
first-fit*	95.8	79.3		1062	301	137–166	592.8	0.886
mod-first-fit*	22.5	835.7		8321	35	601–651	580.2	0.514
best-fit-first*	84.2	50.3		355	262	120–149	594.7	0.909
best-fit-last*	111.7	95.0		348	256	123–150	595.8	0.906
Uniform subpools:								
1-wide	6.5	2.1	0.995	1145	23	157–196	564.2	0.859
2-wide	4.5	1.7	0.996	911	21	154–185	575.0	0.862
3-wide	3.8	1.8	0.995	699	13	166–196	592.4	0.847
4-wide	3.1	1.6	0.995	512	22	157–187	593.1	0.858
2-Level subpools divided at 128 doubleword boundary: (e.g. 2/32 = 64 2-doubleword-wide subpools to 128 then 12 32-doubleword-wide subpools to 512)								
1-32	5.6	1.8	0.995	1052	40	136–163	572.6	0.886
1-32**	2.4	2.1	0.995	1175	28	140–174	572.6	0.881
2-16	4.2	1.6	0.995	830	22	139–164	578.0	0.883
2-32	4.2	1.6	0.996	813	28	140–168	583.2	0.882
2-64	4.5	1.7	0.995	851	31	147–173	594.0	0.872
4-64	3.0	1.6	0.995	407	30	156–184	612.0	0.860
BUDDY SYSTEMS:								
Algorithm	**Mean Items Visited per Request**	**Mean Items Visited per Release**	**Split Rate**	**Join Rate**	**Extend Rate (pages/min)**	**Extended Pages Mean-max**	**Mean Storage Out (pages)**	**Storage Efficiency**
binary	2.00	3.01	0.0080	0.0074	393	372–403	849.9	0.647
binary (no tags)	1.02	29.28	0.0058	0.0052	339	258–293	737.6	0.744
mod-Fibonacci	2.04	3.04	0.0156	0.0147	15	276–308	651.8	0.727

* = minimum fragment left on free list was 5 doublewords.
** = two free lists with maximum size block maintained for each.
dedicated dynamic storage: 500 pages.
mean time between users logging off: 5.7 seconds.
mean requests per second: 1048.
mean requested storage in use: 564.2 pages.
mean number of blocks in use: 17 509.

storage efficiency = (mean requested storage in use)/[(initial storage allocation) + (mean extended storage)].

In addition, for the buddy systems the following metrics were used:

- Split rate: the mean number of splits caused by a free-storage request
- Join rate: the mean number of joins caused by a free-storage return

The YKTVMV and CAMBRIDG simulations were allowed to stabilize for two hours of simulated time and then run for four hours of simulated time. The FRKVM1 simulation was allowed to stabilize for 30 minutes of simulated time and then run for 7.5 hours of simulated time. The stabilization time of the simulation was empirically determined by studying the stabilization of the storage out (i.e., being held). Most algorithms had stabilized (in terms of speed and storage efficiency) a considerable time before the end of the simulation run—a notable exception was modified-first-fit.

The simulation results will be discussed in sections by the general algorithm categories.

VM/SP algorithm. The simulation of the standard VM/SP algorithm confirmed that there was significant

Table 3 YKTVMV simulation results (3033 MP)

Algorithm	Mean Items Visited per Request	Mean Items Visited per Release	Subpool Hit Ratio	Mean Freelist Length	Extend Rate (pages/min)	Extended Pages Mean-max	Mean Storage Out (pages)	Storage Efficiency
standard	69.3	39.4	0.917	1030	282	76–109	802.4	0.887
first-fit	1678.9	1591.1		4574	294	41–72	748.8	0.926
best-fit-last	287.6	197.7		1376	285	10–38	748.8	0.962
best-fit-first	365.0	260.7		1427	231	10–38	748.8	0.963
first-fit*	157.6	124.8		1516	293	67–100	785.8	0.896
mod-first-fit*	80.1	1812.7		11886	196	553–601	776.6	0.567
best-fit-first*	100.8	49.4		350	262	36–68	783.7	0.932
best-fit-last*	113.7	83.6		321	250	39–74	785.8	0.928
Uniform subpools:								
1-wide	7.0	2.8	0.995	1617	15	107–171	748.8	0.856
2-wide	4.6	2.2	0.995	1337	13	105–152	769.3	0.858
3-wide	4.5	2.1	0.995	1215	13	129–168	807.1	0.834
4-wide	4.5	1.9	0.995	1161	8	148–185	835.4	0.817
2-Level subpools divided at 128 doubleword boundary: (e.g. 2/32 = 64 2-doubleword-wide subpools to 128 then 12 32-doubleword-wide subpools to 512)								
1/32	4.8	2.5	0.995	1626	14	75–103	754.2	0.889
1/32**	2.5	2.2	0.995	1632	12	75–106	754.2	0.888
2/16	3.9	2.0	0.996	1397	7	83–113	771.1	0.880
2/32	3.8	2.0	0.996	1405	9	81–110	774.5	0.882
2/64	3.6	2.0	0.996	1381	8	83–112	781.4	0.880
4/64	4.1	1.8	0.996	1133	6	137–168	847.3	0.827

BUDDY SYSTEMS:

Algorithm	Mean Items Visited per Request	Mean Items Visited per Release	Split Rate	Join Rate	Extend Rate (pages/min)	Extended Pages Mean-max	Mean Storage Out (pages)	Storage Efficiency
binary	2.00	3.00	0.0065	0.0054	411	400–445	1140.9	0.641
binary (no tags)	1.02	49.42	0.0063	0.0051	343	278–320	1027.8	0.716
mod-Fibonacci	2.13	3.12	0.0590	0.0576	209	181–219	901.2	0.789

Algorithm	Mean Items Visited per Request	Mean Items Visited per Release	Storage Efficiency
better-fit	14.5	20.8	0.650
leftmost-fit	35.0	41.0	0.928
best-fit-last	284.0	205.4	0.962

* = minimum fragment left on free list was 5 doublewords.
** = two free lists with maximum size block maintained for each.
dedicated dynamic storage: 768 pages.
mean time between users logging off: 14.6 seconds.
mean requests per second: 1034.
mean requested storage in use: 748.8 pages.
mean number of blocks in use: 27 344.

search overhead in the FRKVM1 and YKTVMV systems. In particular, the YKTVMV system had an especially high overhead. The lower subpool hit ratios *vis-à-vis* the CP/67 of the Margolin et al. study are due to the growth in size of system control blocks and buffers over the intervening releases of VM/SP. The mean number of items inspected per request is largely a function of the subpool hit ratio and the mean number of blocks on the free list.

It was surprising that the CAMBRIDG system was performing considerably better than FRKVM1 and YKTVMV. The FRKVM1 3033 UP had approximately 4.5 times the CPU power of the CAMBRIDG processor. However, the total number of blocks searched per second for requests was 32 070 (1048 requests per second × 30.6 items inspected per request) for FRKVM1 and 1070 (198 × 5.4) for CAMBRIDG—a ratio of 30:1. This was only partially accounted for by the

Table 4 CAMBRIDG simulation results (158 UP)

Algorithm	Mean Items Visited per Request	Mean Items Visited per Release	Subpool Hit Ratio	Mean Freelist Length	Extend Rate (pages/min)	Extended Pages Mean-max	Mean Storage Out (pages)	Storage Efficiency
standard	5.4	4.3	0.962	140	34	0.6–9	78.2	0.843
First-fit, best-fit, etc.:								
standard-without-subpooling	54.3	40.2		206	4	0.077–5	76.4	0.848
first-fit	213.3	212.1		640	16	0.5–8	76.4	0.844
best-fit-last	84.0	58.4		258	1	0.008–4	76.4	0.849
best-fit-first	95.9	66.7		263	1	0.008–4	76.4	0.849
first-fit*	33.3	33.0		246	25	1–10	79.7	0.839
mod-first-fit*	7.2	200.2		1034	24	44–53	78.3	0.570
best-fit-first*	35.8	21.7		77	7	0.042–4	79.2	0.849
best-fit-last*	39.4	29.2		77	16	0.077–5	79.2	0.848
Uniform subpools:								
1-wide	2.3	1.5	0.994	318	6	33–59	76.4	0.620
2-wide	1.9	1.3	0.995	259	7	31–52	79.1	0.632
3-wide	1.7	1.2	0.996	221	7	28–45	78.4	0.646
4-wide	1.5	1.2	0.996	181	3	27–41	82.8	0.653
2-Level subpools divided at 128 doubleword boundary: (e.g. 2/32 = 64 2-doubleword-wide subpools to 128 then 12 32-doubleword-wide subpools to 512)								
1-32	1.6	1.3	0.996	281	1	18–27	76.9	0.707
1-32**	1.4	1.3	0.996	290	1	18–30	76.9	0.707
2-16	1.5	1.2	0.997	229	2	21–28	79.2	0.691
2-32	1.4	1.2	0.997	218	1	18–27	79.5	0.707
2-64	1.4	1.2	0.997	215	1	16–22	80.0	0.719
4-64	1.2	1.1	0.997	153	1	16–24	83.7	0.719

BUDDY SYSTEMS: Algorithm	Mean Items Visited per Request	Mean Items Visited per Release	Split Rate	Join Rate	Extend Rate (pages/min)	Extended Pages Mean-max	Mean Storage Out (pages)	Storage Efficiency
binary	2.00	3.01	0.0133	0.0127	47	22–32	103.7	0.683
binary (no tags)	1.03	15.28	0.0139	0.0134	45	21–31	102.7	0.688
mod-Fibonacci	2.04	3.04	0.0224	0.0217	28	27–36	92.0	0.655

* = minimum fragment left on free list was 5 doublewords.
** = two free lists with maximum size block maintained for each.
dedicated dynamic storage: 90 pages.
mean time between users logging off: 73.5 seconds.
mean requests per second: 198.
mean requested storage in use: 76.4 pages.
mean number of blocks in use: 2356.

more favorable CAMBRIDG subpool hit ratio (in the simulation, the subpool hit ratios are 0.962 for CAMBRIDG and 0.943 for FRKVM1). The major factor is that both the request rate and the mean number of items on the free list are roughly proportional to the user load (and therefore the CPU capacity). Consequently, even with a constant subpool hit ratio, this tends to make the overhead (in terms of the number of items inspected per second) of the strategies that search a linked list proportional to the square of the relative system capacity. The fact that supervisor time for free-storage management is not linear with system size is known as a "large system effect," which is undesirable and to be eliminated.

First-fit, modified first-fit, best-fit, and standard VM/SP without subpools. Although it was immediately obvious that none of these algorithms would be competitive in a VM/SP environment, we were surprised by some of the results. Best-fit significantly outperformed first-fit with these distributions. In addition, best-fit-last (i.e., using the last of several equal best fits) was superior to best-fit-first, especially with the larger systems. The standard VM/SP algorithm without subpooling was simulated with the CAMBRIDG distribution and was superior to best-fit.

All of these algorithms were decidedly inferior to the standard VM/SP algorithm. Modified first fit ap-

proaches the standard algorithm in terms of speed but requires over 50 percent additional memory. The VM/SP dynamic storage distribution with its wide range of request sizes and preponderance of requests for smaller-sized blocks presents a difficult environment for modified first-fit (cf. Bays[18]). This algorithm cycles around memory, fragmenting the blocks that are necessary to fill the occasional large request.

In an attempt to reduce the search overhead of these algorithms, Knuth's suggestion[8] to eliminate small fragments (i.e., rounding up the request if the remaining fragment is less than some threshold) was implemented using a threshold of five doublewords. This threshold was derived empirically from simulation experiments. However, we did not increment each request by the doubleword that would be required in VM/SP to keep track of the actual amount of storage given. Therefore, although these results overestimate the storage efficiency in VM/SP, they more closely resemble what might be expected on a system with smaller free-storage granularity and demonstrate the storage overhead inherent in using the threshold.

Using this threshold, best-fit still outperformed first-fit, although the distance between them was greatly reduced. It was discovered that, with the threshold, best-fit-first was superior to best-fit-last.

The mean number of items inspected per release could be significantly reduced for all of these algorithms by adding tags and a size field to each block so that a returned block can be immediately inserted into a doubly linked list. This technique, described in Knuth,[8] would require an additional doubleword in VM/SP and would consequently reduce memory efficiency. The loss due to this internal fragmentation would be about 2360, 17 510, and 27 340 doublewords (i.e., one doubleword for each of the mean number of items being held) for CAMBRIDG, FRKVM1, and YKTVMV, respectively.

Best-fit and first-fit have two advantages:

1. High storage efficiency.
2. The ability to return a block of storage piecemeal. In most of the other strategies studied, a block must be returned as one piece. Although piecemeal release is not required in VM/SP, it might be in other environments.

If piecemeal release is not required, the other advantage, high storage efficiency, comes at such a performance penalty that we doubt these strategies would be attractive on any system with high free-storage activity.

Extended subpooling. Since it was clear that the mean number of items searched in the standard VM/SP algorithm would be reduced if the subpool hit ratio were improved, the next step was to experiment with variations of extended subpooling. First, the subpool coverage was increased from the 1-to-30-doubleword request range of the standard algorithm to 1 to 512 doublewords. The rare request for a block greater than 512 doublewords would cause a search of the free list. The initial work involved subpools of uniform width (i.e., for subpools of width n, the 512-doubleword range is divided into $512/n$ subpools such that the first subpool contains storage of size n and services requests in the range 1 to n, the second subpool contains storage of size $2n$ and services $n+1$ to $2n$, and the kth subpool contains storage of size kn and services $(k-1)n+1$ to kn). The simulation results for subpools of widths 1–4 are given in Tables 2–4 for each of the three systems. Storage efficiency deteriorated with widths greater than four. The uniform-4 (i.e., $n=4$) algorithm was subsequently run on FRKVM1 and YKTVMV. The results are given in the next section.

Nonuniform subpool widths were tried at Yorktown Heights, with the sizes selected to match the measured request frequency, in order to limit internal fragmentation loss. The performance results were very good, but they came at a considerable cost in terms of program complexity. Experience shows that control block and buffer sizes (which largely determine the distribution) are quite dynamic in VM/SP, varying from release to release and also being subject to local modifications. Any closely matched subpooling arrangement would lack robustness, a quality that we wanted to preserve. For this reason, uniform-width subpools were chosen after considering the alternative of tailoring the subpools.

In order to improve the ability of the system to return extended pages after a demand surge, we studied the effect of maintaining extended blocks on a separate subpool for each interval. These extended blocks were then used only if the primary subpool was empty. This approach resulted in better storage efficiency and was used in all of the extended subpooling simulations.

Experiments were done with less severe methods of controlling the subpool inventory than purging them

when a user logs off (this occurred about once every five seconds on FRKVM1). If nothing is done, storage inefficiency results as the subpool inventory grows with demand surges and never shrinks. This phenomenon is explained in detail by Margolin et al. It was found that by time-stamping blocks when they are pushed onto the subpool stack and only releasing the "old" blocks, the speed of the algorithm is improved with no significant loss in storage efficiency. In fact, allowing large blocks to remain in the subpools, if they have been recently used, protects them from being split and reduces this form of external fragmentation. The following procedure was used in all of the extended subpooling simulations:

The two-level approach reduced the external fragmentation loss.

- Whenever a user leaves the system (i.e., logs off) or at least once an hour, a scan is made of all the subpools. (This logging-off was assumed to be a Poisson process; the mean interlog-off time was derived from VM/Monitor data.)
- During this scan all blocks are removed from the subpools dedicated to extended storage and inserted into the free list. Extended pages that are completely contained in the free list are returned to the page pool.
- For the subpools containing nonextended blocks, the total amount of subpool inventory is computed as the subpool is searched. Until the storage contained in the subpool exceeds two pages, an age threshold of 120 seconds is used. After two pages, the threshold is quartered (i.e., 30 seconds). These thresholds have been determined empirically by simulation. Whenever a block is found that has resided in the subpool for this amount of time (whichever is appropriate), that and all older blocks are removed and returned to the free list. After some of these algorithms were installed in VM/SP systems, we studied the relationship between this age threshold and both the subpool hit ratio and storage efficiency. The results of this study are discussed in the next section.

A comparison of the simulation results for uniform-width subpools between the CAMBRIDG and YKTVMV systems is informative. External fragmentation loss in the form of subpool inventory predominates on the smaller CAMBRIDG system, whereas internal fragmentation loss becomes more important on the larger YKTVMV system. For example, of the uniform-width subpools, the four-doubleword width yields the best storage efficiency at CAMBRIDG, and the two-doubleword width is best at YKTVMV. The fact that two-doubleword subpools are slightly more efficient in use of storage than one-doubleword subpools even at YKTVMV is evidence that external fragmentation is still significant. That is, although 20 pages (749 versus 769) less storage were given out with one-doubleword subpools as compared with two-doubleword subpools, two pages (107 versus 105) of additional storage were required because of external fragmentation. This external fragmentation loss was primarily due to inventory in the larger-sized subpools.

This situation led to consideration of the feasibility of a two-width subpooling arrangement that would tend to reduce both the internal and external fragmentation loss and still remain robust. Most of the large requests were found to have short holding times. These are typically I/O buffers. A study of the mean number of items outstanding by size for each of the distributions disclosed that a division at the 128-doubleword size was attractive in that not much storage was being held above this size, and, in addition, it was far enough above the major control block sizes to allow them considerable growth before this condition would change. By using larger-sized subpools above this boundary, we found that the subpool inventory was substantially reduced without a compensatory loss because of internal fragmentation. In the tables and text these two-width subpooling strategies are referred to in the form "L/H," where L equals the width in doublewords of the subpools below the 128-doubleword boundary and H equals the width in doublewords of the subpools above this boundary. On the largest system studied, YKTVMV, several of these algorithms yielded a storage efficiency comparable to the standard VM/SP algorithm with significantly reduced search overhead. The YKTVMV simulation results (see Table 3) show that 1/32 and 2/32 are 14 and 18 times faster respectively than the standard VM/SP algorithm and within one percent of the storage efficiency—with 1/32 actually using one page less.

One of the effects of the two-level approach was to reduce external fragmentation loss to the point where

internal fragmentation predominates. Those algorithms that used the minimum-width subpool (i.e., one doubleword) below the boundary were superior in terms of memory efficiency (within this class of algorithm) on the two largest systems (FRKVM1 and YKTVMV). The fact that their efficiency increased with system size is an indication that internal fragmentation becomes increasingly important as the number of blocks in use increases.

In a buddy system, storage is allocated in subpools of varying size.

On systems that extend frequently, the fact that extended blocks are ordered on the front of the free list impacts the performance of any algorithm (such as extended subpooling) that is an extension of the standard VM/SP algorithm. This is because of the preference for items within the dedicated free-storage block (i.e., nonextended) when the free list is searched (see earlier discussion on background). Therefore, the simulation of one of the best extended subpooling algorithms was modified so that two free lists were maintained: one each for extended and nonextended blocks. In addition, the size of the largest block on each free list and the number of these on the free list was maintained. In this way, the search of a free list could be completely avoided if failure was certain. In addition, first-fit was used on the extended free list. The results indicated significant further improvement of the speed of these algorithms on the larger systems. On YKTVMV the 1/32 strategy with two free lists had one-half the search overhead of the "standard" 1/32 with essentially the same storage efficiency (a difference of less than one page). On FRKVM1 the search overhead was slightly less than one-half, and four additional pages were required (an increase of less than one percent).

Finally, the effect of splitting a block from a larger subpool to satisfy a request for an empty, smaller subpool if the system would otherwise have to extend was studied. This was being done in the standard VM/SP algorithm. The simulation results indicated that this splitting was not worthwhile as it resulted in slightly less speed and storage efficiency because of increased external fragmentation. With the 1/32 strategy and the YKTVMV parameters, the overhead increased to 5.5 items inspected per request (from 4.8), and the number of extended pages increased to 77 (from 75). Attempts to improve this outcome by setting a minimum size on the fragment left by the split were unsuccessful.

Buddy systems. In a buddy system, storage is allocated in subpools of varying size (e.g., powers-of-two in the binary buddy system). Initially all storage is allocated in the largest subpool or in a large contiguous block separate from the subpools. All requests are rounded up to the nearest subpool boundary. If a requested subpool is empty, the next larger subpool is checked, and if it is not empty, a block is taken and split into buddies, one of which is used to satisfy the initial request. If the next larger subpool is empty, the search/split logic is applied recursively until the request is satisfied. Upon release, if the buddy of the block is available (i.e., not in use), they are joined and placed in the larger subpool. This joining continues until the largest size is reached or a buddy is found to be in use.

In the binary buddy system, buddies are always the same size. In other buddy systems such as the Fibonacci and generalized Fibonacci, this is not the case. This inequality not only tends to complicate the algorithm, but, as the simulations demonstrated, it can also increase the external fragmentation. In nonbinary systems, an active subpool with long hold times can cause frequent splitting of its neighbor into its own size and a potentially unpopular size. This combination will build up large unusable subpool inventories and reduce the storage efficiency of the system. We call this the "sawdust phenomenon."

Buddy systems were among the fastest strategies that we studied, all of them having close to the minimum search overhead. The following buddy systems were simulated.

Binary buddy with tags. Knuth[8] recommends a "tagged" buddy system in which a tag field is kept with each block of storage. This tag field indicates whether the block of storage is in use or free. When a block is returned, one need only check the tag of the proper adjacent block of storage to decide if the buddy is free. In VM/SP the tag requires an additional doubleword for each block since storage is given out

Table 5 Modified Fibonacci buddy characteristics

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
s	1	2	3	4	5	7	10	14	19	26	36	50	69	95	131	181	250	345	476	512
sl	*	1	1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	11
sh	*	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
jl	*	1	1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	11	*
jh	*	5	6	7	8	9	10	11	12	13	14	15	16	17	18	*	*	*	*	*

*means that the particular case could not happen.

in doubleword units. The increased storage requirements caused by this situation might not be as severe on other systems with a smaller storage granularity.

Binary buddy without tags. This algorithm was the same as above except that tags were not used, and therefore, on release a search had to be made for the buddy. The increased storage efficiency was notable on the large systems. However, in our opinion, this improvement was more than offset by the increased search overhead on returns.

Modified Fibonacci buddy. The Fibonacci buddy systems were recommended[8] as a possible solution to the internal fragmentation characteristics of the binary buddy. We attempted to tailor the Fibonacci buddy system to our distributions by using a modified Fibonacci sequence[20] as follows:

$$F(1) = 1$$
$$F(2) = 1$$
$$F(3) = 1$$
$$F(4) = 1$$
$$F(5) = 1$$
$$F(n) = F(n-1) + F(n-4) \text{ (approximately)}$$

Since the first five terms of this sequence are the same size, they are combined into one subpool. Some adjustment is necessary to make the subpool sizes come out so that the largest subpool is one page (512 doublewords). Table 5 was used in determining how to split and join subpools. In this table "*n*" is the subpool number, "*s*" is the number of doublewords in the subpool, "sl" is the subpool in which to place the lower buddy at split time, "sh" is the subpool in which to place the upper buddy at split time, "jl" is the subpool to search for the lower buddy at join time, and "jh" is the subpool to search for the upper buddy at join time.

This modified Fibonacci buddy gave mixed results. On the largest system, YKTVMV, it was significantly superior to both forms of the binary buddy. The "sawdust phenomenon" was evident with the FRKVM1 and CAMBRIDG distributions and was the reason this strategy did not perform as well there as at YKTVMV. We think that this lack of robustness of the Fibonacci buddy systems could be a significant problem in an operating system with control block structures that change as frequently as those in VM/SP. On systems with a fixed, well-defined storage request distribution, it might be possible to "tune" the modified Fibonacci systems to give good storage efficiency combined with low CPU overhead. But a simpler subpooling algorithm will probably give comparable results with less complexity and more robustness.

Better-fit and leftmost-fit. After this study was completed, C. J. Stephenson informed us of two new algorithms, better-fit and leftmost-fit, that he had devised using a "cartesian" tree. A detailed description of cartesian trees and the storage allocation strategies based on them is in Stephenson.[26] When used for storage allocation, a cartesian tree has the following properties for any node S:

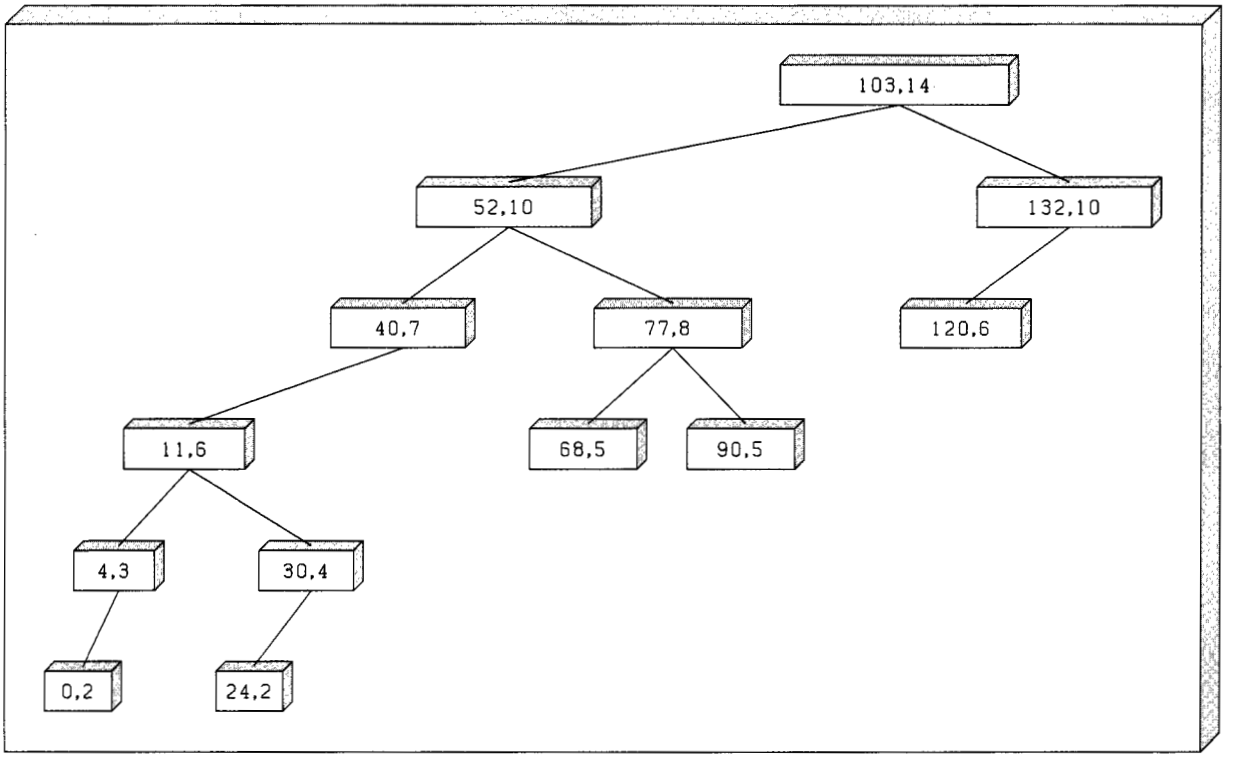
1. Addresses of left descendants (if any) < address of S > address of right descendants (if any)
2. Length of left son (if any) ≤ length of S ≥ length of right son (if any)

Figure 1 is an example of a cartesian tree as it might appear in a dynamic storage allocation application. For descriptive purposes, each node contains a tuple of the form *a*,*s* where

a = address of storage block
s = size of storage block

(Stephenson points out that in practice it is often preferable to have the size of each node contained in its parent, i.e., in the same place as its address.) The "anchor" or head of the free-storage list points to the root (103,14) of this tree.

Figure 1 Example of a cartesian tree used for dynamic storage allocation



The better-fit strategy selects a node by descending the tree, from the root, so that at each decision point the better-fitting son is chosen. The descent stops when both sons are too short or nonexistent.

Leftmost-fit selects the leftmost node of sufficient length. It is identical to first-fit in terms of the storage that is allocated.

In order to test these strategies in the VM/SP environment, simulations were run using the YKTVMV parameters. Stephenson provided the algorithms that were then adapted to the Pascal simulator. The algorithms were not written specifically for the VM/SP environment and therefore were given sufficient storage to avoid extending. We also ran best-fit in a nonextend mode so that the difference between the extend and nonextend modes would be measured.

Initial simulation results indicated that leftmost-fit performed well but that better-fit suffered from severe external fragmentation. The three strategies (best-fit, better-fit, and leftmost-fit) were then modified so that, after the simulation had stabilized, the fragment size left after satisfying a request was counted by size. The results are given in Table 6. It is clear that the "winning" node in better-fit causes more fragmentation than in best-fit or leftmost-fit. The VM/SP distribution seems to be almost as pathological for this strategy as for modified first-fit. Stephenson has found that better-fit works well with other distributions, and therefore it is of potential interest in applications.

The relative speed of leftmost-fit makes this algorithm a good strategy to use "behind" subpooling. If this were done in VM/SP, for example, the extended subpooling strategies would all have mean-items-visited values that are less than 2.0. Perhaps more important for machines with a cache, leftmost-fit disturbs the cache significantly less than the traditional linked-list strategies.

System results

Before the simulation study was completed, the uniform-4-wide algorithm was run for two months on FRKVM1 and subsequently for a shorter period on

Table 6 Fragments left by best-fit, better-fit, and leftmost-fit (YKTVMV distribution)

Size of Fragment Left	Cumulative % of Total Fragments		
	Best-Fit	Better-Fit	First-Fit and Leftmost-Fit
0	88.774	49.278	65.354
1	93.735	62.687	77.001
2	95.586	68.550	82.890
3	96.734	72.310	86.692
4	97.590	75.357	89.595
5	98.060	77.858	91.720
6	98.329	79.676	93.229
7	98.562	81.308	94.596
8	98.789	82.769	95.542
9	98.961	84.687	96.350
10	99.069	86.961	96.758
20	99.532	93.026	98.073
30	99.675	95.725	98.511
40	99.750	97.224	98.816
50	99.777	98.113	98.980
60	99.796	98.731	99.096
70	99.810	99.090	99.200
80	99.819	99.333	99.272
90	99.827	99.506	99.333
100	99.832	99.628	99.383
200	99.900	99.950	99.647
300	99.939	99.980	99.799
400	99.946	99.986	99.850
500	99.954	99.993	99.886

Table 7 Standard VM/370 versus 2/32 at YKTVMV

	Base System	Modified System	Percent Improvement
System CPU	82.8	78.2	5.6 reduction
Problem CPU	67.8	74.8	10.3 increase
System/prob	1.221	1.045	14.4 decrease
Free lock spin (percent of elapsed time)	1.1	0.1	
free lock hold (percent of elapsed time)	14.8	4.5	69.6 decrease

YKTVMV. After the advantages of two-level subpooling became evident, it was replaced with the 2/32 algorithm (i.e., two-doubleword-wide subpools up to the 128-doubleword boundary and then 32-doubleword-wide subpools for those sizes above this boundary) on both systems. This replacement gave us an opportunity to test the predictive ability of the simulator with two algorithms.

Hardware and software monitoring at Franklin Lakes and Yorktown Heights indicated a reduction in the supervisor state CPU utilization to 4–5 percent from the previous 15–20 percent for both of these algorithms. At FRKVM1 the subpool hit ratios were monitored from 8:30 AM to 4:30 PM (the period of heaviest daily activity) and found to be consistently within +/− 0.001 of the predicted values on a daily basis.

The simulation results indicated that on FRKVM1 the 2/32 algorithm would require 0.974 of the storage needed by uniform-4 wide. The observed value was 0.973. This comparison was not made on YKTVMV because the uniform-4 algorithm was run with a free-storage "trap" (a method of "trapping" dynamic storage release violations by appending extra storage containing size and requestor information to each request) which was not used with 2/32.

At YKTVMV a comparison was made of the standard VM/SP algorithm with 2/32. Evaluation was done by comparing software monitor data from the same hour of the same day of the week for the base and modified systems. The results are shown in Table 7.

The most valid measure of the overall value is the 10.3 percent increase in virtual time. The 5.6 percent reduction in supervisor time understates the value of the change. The decreased supervisor time allows more virtual time, which in turn increases supervisor time because of services required.

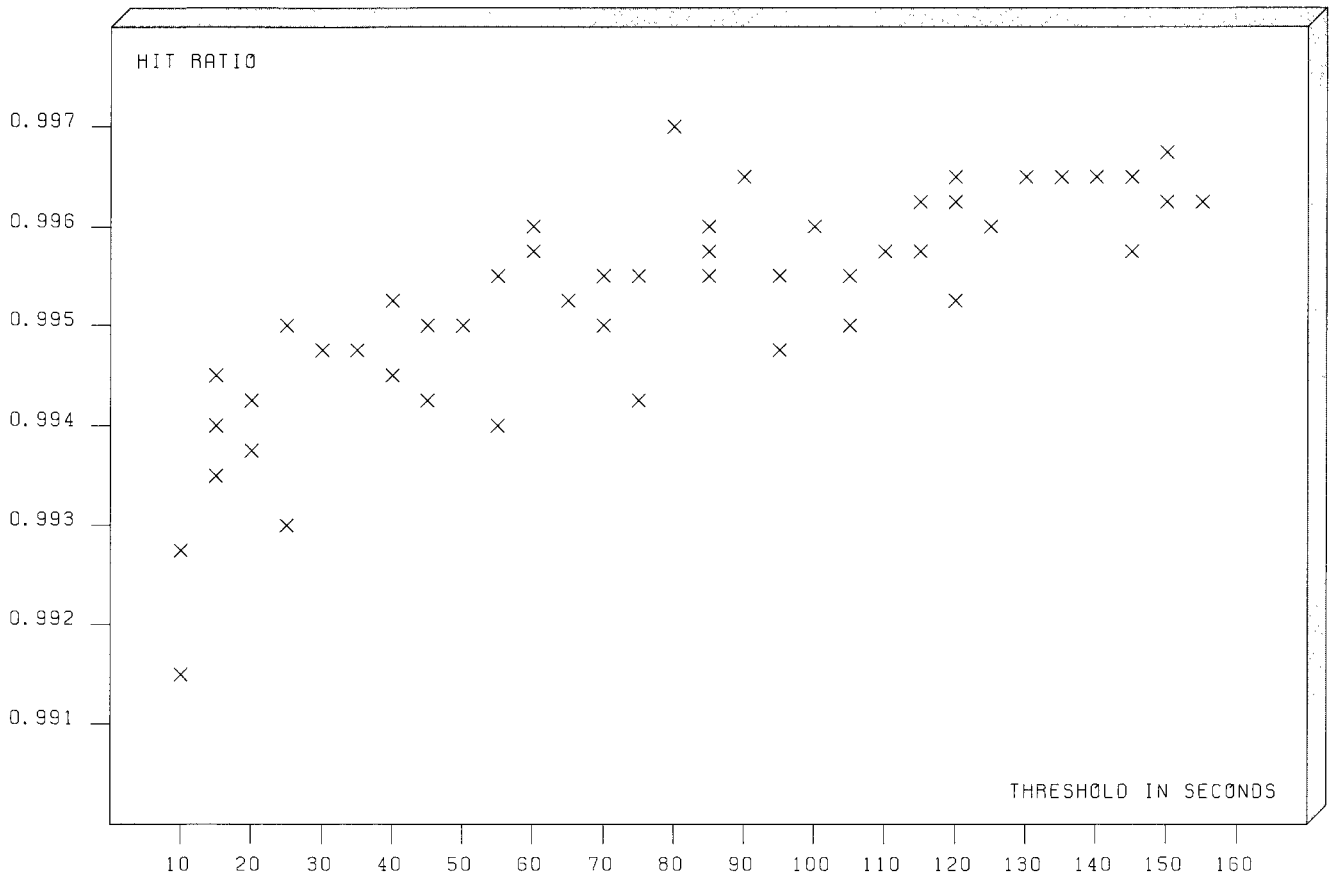
The 2/32 strategy results in free-storage management being reduced from 14.8 percent to 4.5 percent of elapsed time. This in turn allows about ten percent more virtual time to be given to the users of the system.

The System/Prob ratio shows the supervisor time needed to support one unit of virtual time. Thus, the supervisor time per unit of useful work has been reduced by 14.4 percent.

During the study a model was created to relate lock holding probability to lock spin probability. The model was useful because the VM/Monitor is able to report lock spin time, but not lock holding time. Lock holding is another way to measure the time spent doing free-storage management, because that is the only use of the lock.

The simulator predicted that the 2/32 strategy would require 0.6 percent more storage than standard VM/SP on YKTVMV. During the period of the test at YKTVMV, the 2/32 strategy used an average of 0.6 percent less storage than standard VM/SP. This result (combined with the FRKVM1 results) is evidence that

Figure 2 Subpool hit ratio as a function of subpool release time threshold at FRKVM1

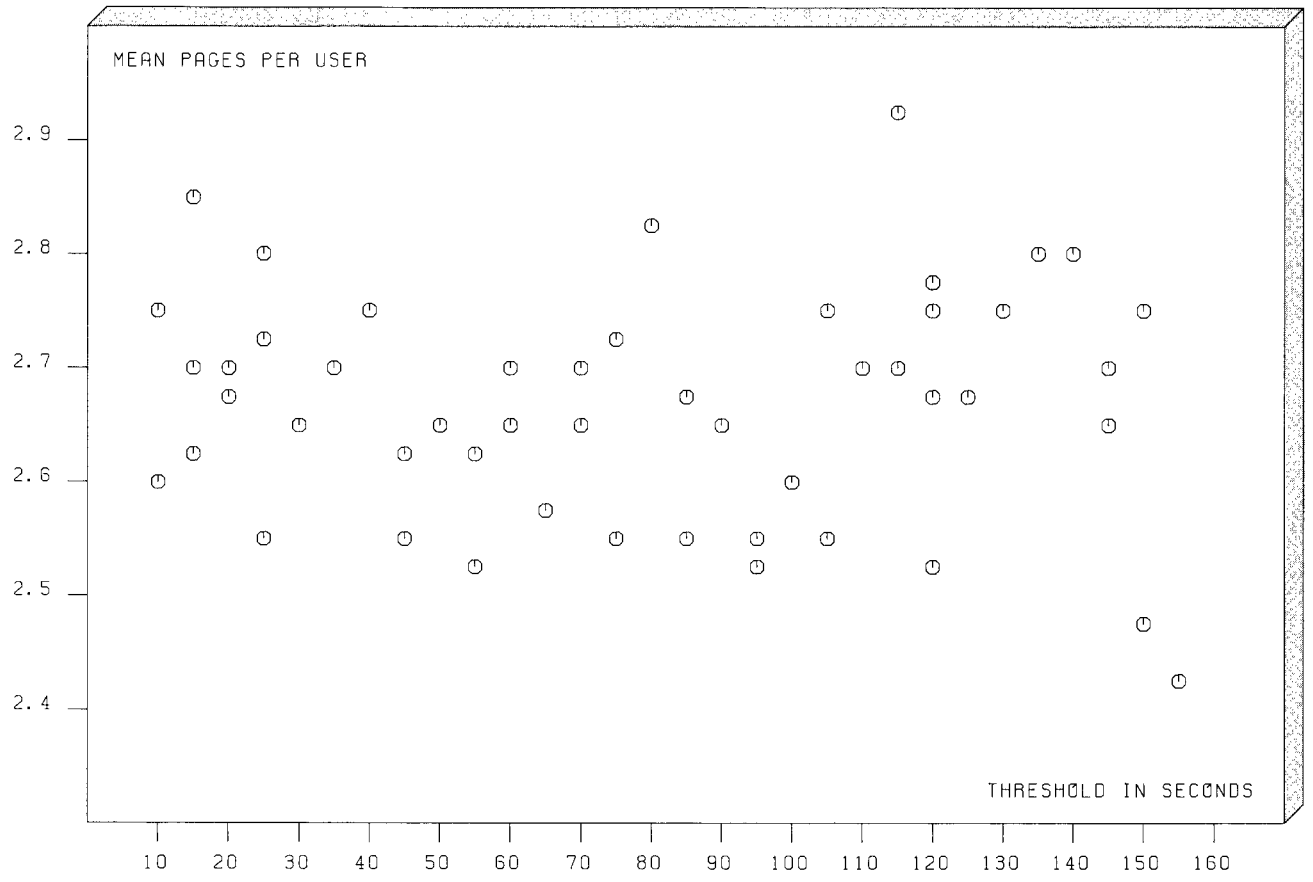


this method of simulation can predict the relative storage efficiency of allocation strategies with reasonable accuracy.

Finally, the speed-storage tradeoff involved in the subpool release time threshold was studied. This time was used as the age criterion for removing blocks from the subpools whenever a user logged off (except that all extended blocks were always removed). After the 2/32 algorithm was installed on FRKVM1, the time threshold was varied and plotted to show the relationship to the subpool hit ratio and the mean number of pages required per user [= (dedicated pages + mean extended pages)/(mean number of logged-on users)]. The results are shown in Figures 2 and 3. Each point on the graph is one day's observation. Note that although the range of subpool hit ratios is narrow, the simulation study indicated that very small changes in the hit ratio adversely affected the mean search overhead due to the size of the free list. It is apparent that the subpool hit ratio is affected by the release threshold. The correlation coefficient is 0.76. However, storage efficiency does not appear to have any relationship to this threshold over the time range studied: the correlation coefficient is −0.08.

The simulation study indicated that large threshold values will result in serious external fragmentation in the form of large subpool inventories. Also, it was observed that hardware and load anomalies occasionally cause a transient demand surge for a specific storage size (e.g., the storage associated with an I/O event), and large threshold values hinder the ability of the system to reuse this storage. This observation suggests that the threshold should be set at the point where the subpool hit ratio starts to flatten out. Such a setting results in very low search overhead and still allows the system to recover from demand surges in a reasonable amount of time.

Figure 3 Mean storage per logged user as a function of subpool release time threshold at FRKVM1



Conclusions

Empirical results indicate that simplifying distribution assumptions about interarrival and holding times based on estimated means can be used with high predictive ability in the simulation of dynamic storage systems. The traditional best-fit and first-fit strategies, which are based on the searching of a linked list of available blocks, are too slow for large systems with the dynamic storage characteristics of VM/SP. The standard VM/SP algorithm was found to have high search overhead on the larger systems. Indeed, there is evidence that this search overhead increases approximately in relation to the square of the relative system capacity. The buddy systems, which have been popular in the recent literature, were among the fastest strategies studied, though severely handicapped by storage inefficiency. An extended subpooling strategy was described that is as fast as the buddy systems with superior storage efficiency.

As a result of this study, a generalized two-level subpooling algorithm (nominally 2/32) was incorporated in Release 2 of the HPO (High Performance Option) for VM/SP.

Acknowledgment

The authors thank C. J. Stephenson for many helpful discussions and several key contributions in the study of the traditional (first-fit and best-fit) free-storage algorithms. We thank Joe Reardon of the IBM Cambridge Scientific Center for his assistance in getting parameters from their system. Lastly, we are indebted to the anonymous referees whose suggestions have improved the paper.

Appendix: System dynamic storage parameters

Table 8 FRKVM1 parameters (Interarrival and holding times in seconds)

Size	IAT	HT	Size	IAT	HT	Size	IAT	HT	Size	IAT	HT	Size	IAT	HT
1	0.320	258.425	67	29.671	0.653	133	42.858	1.415	199	122.728	0.010	265	270.000	0.010
2	0.0655	125.260	68	38.029	0.837	134	42.858	0.943	200	61.364	1.350	266	16.072	0.354
3	0.307	920.525	69	36.000	0.396	135	35.065	343.251	201	64.286	0.010	267	385.715	0.010
4	0.305	320.293	70	45.000	435.015	136	42.858	1.415	202	51.924	1.143	268	31.396	0.010
5	0.625	277.229	71	45.763	0.504	137	0.565	0.107	203	50.000	0.010	269	207.693	0.010
6	0.208	165.560	72	49.091	0.540	138	48.215	0.531	204	25.472	0.010	270	450.000	0.010
7	0.0326	14.890	73	27.836	0.919	139	41.539	0.914	205	100.000	100.000	271	142.106	1.564
8	1.127	144.338	74	4.259	0.094	140	34.178	1.128	206	96.429	3.183	272	122.728	0.010
9	0.00312	0.343	75	22.500	233.505	141	49.091	241.086	207	207.693	2.285	273	60.000	60.000
10	0.00282	0.158	76	36.487	0.010	142	35.527	1.990	208	41.539	745.367	274	2700.000	0.010
11	0.00631	0.395	77	23.077	0.254	143	18.000	637.992	209	150.000	1.650	276	4.012	0.490
12	0.0165	1.678	78	48.215	0.010	144	25.715	0.566	210	36.487	1.205	277	8.710	1.647
13	0.698	0.877	79	22.500	0.743	145	40.910	1.350	211	207.693	0.010	278	158.824	0.010
14	0.253	1.877	80	4.624	25.738	146	38.029	0.837	212	36.987	0.010	279	6.068	0.067
15	0.491	35.754	81	31.765	0.010	147	38.029	0.837	213	7.827	0.087	280	50.944	1.121
16	0.225	671.280	82	43.549	0.480	148	40.910	1.800	214	4.252	0.047	281	540.000	0.010
17	0.387	4.986	83	3.431	0.151	149	32.927	0.725	215	42.188	0.010	282	450.000	0.010
18	0.0475	4.480	84	5.745	1632.512	150	36.000	2.412	216	103.847	0.010	283	207.693	0.010
19	0.198	8.426	85	41.539	60.937	151	71.053	0.782	217	128.572	0.010	284	300.000	0.010
20	0.278	77.406	86	39.706	1.311	152	84.375	1.857	218	48.215	0.010	285	675.000	0.010
21	6.459	6.459	87	36.987	0.814	153	31.396	0.691	219	22.690	0.500	286	81.819	115.446
22	5.047	0.010	88	21.600	0.010	154	29.671	1.306	220	2.594	1.326	287	2700.000	0.010
23	10.305	12.706	89	25.715	0.283	155	40.910	1.800	221	9.061	140.138	288	4.531	1.160
24	0.187	743.065	90	45.000	103.995	156	22.132	417.040	222	60.000	0.010	289	540.000	5.940
25	1.346	125.637	91	44.263	0.010	157	45.000	1.485	223	65.854	0.010	290	675.000	0.010
26	20.000	0.010	92	0.616	0.055	158	44.263	1.948	224	45.763	0.010	291	540.000	0.010
27	67.500	0.010	93	30.000	0.330	159	41.539	0.914	225	108.000	0.010	292	540.000	0.010
28	1.422	0.047	94	25.472	0.281	160	50.944	54.357	226	38.029	0.010	293	2700.000	0.010
29	0.579	0.058	95	61.364	10.248	161	20.931	0.231	227	7.606	0.010	294	192.858	0.010
30	0.554	45.978	96	36.987	1.221	162	52.942	1.748	228	2.306	0.077	295	2700.000	0.010
31	12.108	540.811	97	29.348	0.323	163	61.364	2.025	229	158.824	0.010	296	1350.000	14.850
32	23.894	0.010	98	48.215	0.010	164	50.000	0.550	230	15.607	0.344	297	2700.000	0.010
33	51.923	0.010	99	1.193	0.027	165	44.263	1.948	231	270.000	0.010	298	2700.000	0.010
34	87.097	0.010	100	36.987	75.194	166	57.447	3.218	232	22.132	0.487	299	117.392	146.035
35	24.545	532.096	101	8.971	0.297	167	77.143	2.546	233	57.447	0.010	300	900.000	0.010
36	0.232	2.110	102	36.987	0.407	168	84.375	1.857	234	49.091	544.369	301	45.000	0.010
37	81.818	0.010	103	33.750	0.372	169	21.775	518.226	235	72.973	0.803	302	2700.000	0.010
38	0.294	0.082	104	15.607	65.550	170	41.539	0.457	236	51.924	0.572	303	2700.000	0.010
39	21.774	1000.655	105	23.077	0.508	171	60.000	0.660	237	29.033	0.010	304	207.693	2.285
40	0.319	2.660	106	14.063	0.155	172	36.487	36.487	238	300.000	0.010	312	128.572	132.815
41	28.723	0.316	107	27.552	0.304	173	54.000	1.188	239	3.948	0.569	314	1.790	0.120
42	39.130	161.726	108	34.178	0.010	174	42.858	0.010	240	8.518	1.610	325	135.000	48.060
43	28.421	0.010	109	40.299	0.010	175	93.104	0.010	241	90.000	0.990	329	2700.000	0.010
44	34.615	0.010	110	5.649	0.379	176	57.447	2.528	242	5.379	0.237	333	2700.000	0.010
45	33.750	241.515	111	43.549	1.438	177	87.097	1.917	243	9.408	0.207	338	142.106	96.348
46	103.846	0.010	112	31.035	0.010	178	158.824	0.010	244	150.000	0.010	351	135.000	2.970
47	3.121	530.186	113	31.035	0.683	179	84.375	5.654	245	42.858	0.010	364	135.000	0.010
48	41.538	0.010	114	45.763	0.504	180	75.000	3.300	246	50.944	0.010	376	29.671	0.327
49	34.615	0.381	115	36.987	0.407	181	62.791	1.382	247	2.971	25.281	377	135.000	1.485
50	30.000	543.330	116	33.334	0.010	182	33.750	605.239	248	45.000	0.010	390	142.106	0.010
51	27.835	1.225	117	16.266	465.360	183	39.706	0.437	249	57.447	0.010	401	7.737	0.086
52	31.765	1.398	118	39.706	0.874	184	64.286	1.415	250	4.405	1.128	403	142.106	108.995
53	30.337	0.334	119	9.061	0.299	185	71.053	0.782	251	14.674	0.646	416	150.000	1305.000
54	65.854	2.173	120	25.000	0.550	186	84.375	0.010	252	52.942	0.010	429	207.693	990.070
55	26.471	40.871	121	54.000	1.188	187	112.500	3.713	253	21.952	0.242	431	4.116	0.231
56	2.970	0.131	122	24.546	0.810	188	50.000	0.010	254	96.429	0.010	442	245.455	1161.737
57	41.538	0.457	123	40.299	0.887	189	56.250	1.238	255	2.622	1.340	451	0.292	0.163
58	0.320	0.010	124	48.215	1.592	190	64.286	0.708	256	12.108	0.010	455	300.000	563.400
59	67.500	0.743	125	24.771	0.545	191	96.429	1.061	257	7.989	0.176	468	337.500	1019.925
60	27.551	71.027	126	22.500	0.990	192	42.858	0.472	258	192.858	0.010	481	10.113	22.925
61	32.530	0.010	127	18.494	0.814	193	58.696	0.646	259	93.104	0.010	494	900.000	1479.600
62	19.853	0.437	128	4.937	0.109	194	61.364	0.010	260	65.854	337.303	501	2700.000	0.010
63	35.526	0.010	129	44.263	0.010	195	33.750	446.614	261	72.973	0.010	507	2700.000	9871.200
64	33.750	0.371	130	12.386	374.310	196	142.106	4.690	262	8.360	0.010	509	11.490	0.253
65	3.277	19.552	131	20.770	1.392	197	135.000	1.485	263	2.316	0.051	512	900.000	5799.600
66	61.364	0.010	132	56.250	1.857	198	108.000	0.010	264	67.500	0.743			

Table 9 YKTVMV parameters

Size	IAT	HT	Size	IAT	HT	Size	IAT	HT	Size	IAT	HT	Size	IAT	HT
1	0.111	235.114	74	1.521	0.077	147	36.000	0.010	220	3.282	5.579	293	171.429	0.010
2	0.071	273.841	75	56.250	0.010	148	43.374	0.010	221	0.583	0.156	294	124.138	0.010
3	0.146	162.205	76	109.091	0.010	149	25.532	0.010	222	2.215	0.149	295	514.286	0.010
4	0.160	36.054	77	65.455	0.010	150	35.644	0.606	223	76.596	0.010	296	211.765	0.010
5	0.257	869.713	78	65.455	0.010	151	35.644	0.010	224	211.765	0.010	297	240.000	240.000
6	0.089	199.510	79	50.705	204.491	152	25.532	0.010	225	211.765	211.765	298	112.500	0.010
7	0.029	21.422	80	45.570	94.922	153	8.675	155.278	226	73.470	0.010	299	360.000	0.010
8	0.369	186.698	81	30.253	992.784	154	31.579	0.010	227	36.000	0.010	300	171.429	0.010
9	0.081	36.310	82	59.017	0.010	155	70.589	1.200	228	6.041	0.200	301	26.278	0.010
10	0.00157	4.742	83	1.209	0.122	156	92.308	0.010	229	9.231	0.157	302	240.000	0.010
11	0.00655	0.089	84	51.429	0.010	157	43.903	0.747	230	30.000	0.510	303	720.000	0.010
12	0.041	16.623	85	40.450	2.023	158	92.308	0.010	231	50.000	0.010	304	257.143	0.010
13	0.296	1.230	86	44.445	3.689	159	67.925	1.155	232	25.532	0.010	305	81.819	0.010
14	0.423	13.527	87	94.737	3.127	160	1.117	1.136	233	124.138	0.010	306	276.923	0.010
15	0.080	4.119	88	120.000	2.040	161	37.114	0.010	234	55.385	0.010	307	163.637	0.010
16	0.420	1240.163	89	48.000	0.816	162	43.903	395.869	235	90.000	0.010	308	300.000	0.010
17	0.153	11.882	90	23.842	2175.902	163	62.069	0.010	236	49.316	0.839	309	257.143	0.010
18	0.299	63.398	91	85.715	0.010	164	72.000	0.010	237	75.000	0.010	310	327.273	0.010
19	0.109	4.506	92	0.880	0.044	165	29.269	0.498	238	16.290	0.277	311	257.143	0.010
20	0.194	31.643	93	120.000	2.040	166	92.308	92.308	239	3.472	0.289	312	514.286	0.010
21	11.321	51.317	94	52.942	0.900	167	120.000	0.010	240	15.063	0.498	314	3.232	0.055
22	8.824	17.938	95	78.261	5.244	168	105.883	0.010	241	47.369	0.010	315	1800.000	1800.000
23	16.438	69.321	96	50.705	0.010	169	133.334	0.010	242	21.053	0.010	316	600.000	0.010
24	0.284	583.533	97	31.579	2.622	170	50.705	1.674	243	64.286	64.286	317	1800.000	0.010
25	5.136	3044.506	98	87.805	0.010	171	51.429	612.875	244	70.589	0.010	318	600.000	0.010
26	6.883	0.010	99	24.162	1793.163	172	102.858	0.010	245	54.546	0.010	319	240.000	0.010
27	6.486	0.538	100	42.353	43.073	173	51.429	0.010	246	47.369	2.369	320	900.000	0.010
28	5.070	0.010	101	2.488	0.207	174	45.000	0.010	247	1.822	0.214	321	3600.000	0.010
29	0.406	0.034	102	53.732	0.914	175	29.509	0.010	248	26.667	0.454	322	1200.000	0.010
30	1.479	162.055	103	53.732	0.914	176	54.546	0.928	249	60.000	0.010	323	100.000	0.010
31	0.038	16.163	104	41.861	0.712	177	48.649	0.010	250	8.552	3.139	327	3600.000	0.010
32	34.615	2.873	105	69.231	3.462	178	61.017	0.010	251	102.858	0.010	328	600.000	0.010
33	22.086	0.729	106	92.308	0.010	179	105.883	0.010	252	45.000	45.000	329	600.000	0.010
34	45.000	0.010	107	48.649	0.828	180	73.470	383.290	253	37.114	0.010	330	600.000	0.010
35	0.372	81.556	108	33.645	1635.712	181	76.596	0.010	254	3.766	3.326	331	450.000	0.010
36	20.112	100.901	109	52.942	0.900	182	163.637	0.010	255	0.788	0.289	332	900.000	0.010
37	1.468	1.639	110	6.197	0.106	183	144.000	0.010	256	0.851	0.228	333	276.923	0.010
38	0.582	0.029	111	102.858	1.749	184	62.069	1.056	257	7.469	0.010	334	1200.000	0.010
39	16.514	3397.707	112	63.158	1.074	185	156.522	0.010	258	83.721	0.010	335	900.000	0.010
40	64.286	1842.879	113	64.286	1.093	186	87.805	1.493	259	72.000	0.010	336	720.000	0.010
41	48.649	0.010	114	80.000	1.360	187	73.470	1.249	260	211.765	0.010	337	360.000	0.010
42	61.017	2.014	115	100.000	100.000	188	35.644	0.010	261	67.925	118.868	338	3600.000	0.010
43	50.000	50.000	116	40.000	0.010	189	144.000	1000.800	262	83.721	0.010	339	1200.000	0.010
44	63.158	1.074	117	40.000	2264.680	190	102.858	0.010	263	13.954	0.010	340	3600.000	0.010
45	55.385	1047.711	118	33.963	0.578	191	80.000	1.360	264	171.429	5.658	342	3600.000	0.010
46	52.174	0.010	119	5.599	0.185	192	87.805	0.010	265	11.689	0.386	345	3600.000	0.010
47	0.956	245.050	120	29.509	0.502	193	133.334	0.010	266	32.143	0.547	346	240.000	0.010
48	17.648	12862.935	121	67.925	2.242	194	150.000	0.010	267	50.705	0.010	347	52.942	0.010
49	27.693	384.453	122	59.017	1.948	195	87.805	0.010	268	29.509	0.010	348	3600.000	0.010
50	29.033	167.429	123	64.286	1.093	196	19.673	0.010	269	257.143	0.010	350	94.737	0.010
51	38.298	0.651	124	52.174	0.010	197	45.570	0.010	270	163.637	204.546	352	1800.000	0.010
52	67.925	3.397	125	81.819	1.391	198	94.737	552.600	271	240.000	0.010	353	900.000	0.010
53	80.000	1.360	126	45.570	2425.079	199	54.546	0.010	272	80.000	0.010	355	720.000	0.010
54	133.334	0.010	127	31.859	1.052	200	55.385	0.010	273	53.732	1.774	356	3600.000	0.010
55	42.353	58.575	128	2.418	0.201	201	225.000	0.010	274	128.572	0.010	360	1800.000	1800.000
56	1.930	0.033	129	21.053	0.010	202	50.000	0.010	275	16.745	0.285	361	3600.000	0.010
57	60.000	1.020	130	6.991	0.469	203	171.429	0.010	276	3.374	0.281	362	3600.000	0.010
58	22.642	0.010	131	44.445	1.467	204	120.000	0.010	277	15.190	0.502	364	3600.000	0.010
59	45.570	0.010	132	80.000	0.010	205	150.000	300.000	278	257.143	0.010	365	3600.000	0.010
60	40.450	161.798	133	63.158	1.074	206	257.143	0.010	279	327.273	0.010	366	360.000	0.010
61	24.162	0.010	134	116.129	0.010	207	65.455	363.273	280	63.158	0.010	367	1800.000	0.010
62	0.149	0.325	135	76.596	3658.749	208	156.522	0.010	281	360.000	0.010	370	105.883	0.010
63	51.429	55.697	136	81.819	0.010	209	25.900	0.010	282	100.000	0.010	377	720.000	0.010
64	67.925	409.789	137	0.192	0.058	210	39.561	0.010	283	133.334	0.010	382	900.000	0.010
65	1.311	1.442	138	36.735	0.625	211	62.069	0.010	284	109.091	1.855	401	52.174	0.010
66	58.065	0.010	139	60.000	0.010	212	76.596	0.010	285	36.735	0.010	431	514.286	0.010
67	92.308	1.570	140	27.273	0.900	213	1.995	0.234	286	28.800	0.490	451	2.755	0.138
68	94.737	0.010	141	46.154	0.785	214	36.735	0.010	287	102.858	0.010	481	0.381	0.032
69	29.269	1.464	142	41.861	0.010	215	276.923	0.010	288	87.805	0.010	483	18.368	0.607
70	27.907	29.777	143	102.858	1.749	216	90.000	361.530	289	4.187	1.466	501	2.757	0.010
71	85.715	1.458	144	90.000	2169.000	217	138.462	0.010	290	180.000	0.010	512	5.715	22.955
72	26.667	162.214	145	0.680	0.012	218	37.114	0.010	291	211.765	0.010			
73	9.891	5212.751	146	73.470	0.010	219	4.206	0.072	292	120.000	0.010			

Table 10 CAMBRIDG parameters

Size	IAT	HT	Size	IAT	HT	Size	IAT	HT	Size	IAT	HT	Size	IAT	HT
1	1.031	33.633	57	600.000	600.000	132	360.000	0.010	234	514.286	531.258	470	600.000	0.010
2	0.471	52.478	58	189.474	0.010	135	900.000	3480.300	236	720.000	0.010	475	44.445	0.010
3	2.200	208.516	59	128.572	0.010	137	1.177	0.039	237	300.000	0.010	481	4.134	0.207
4	2.007	42.608	60	276.924	0.010	139	92.308	0.010	238	400.000	0.010	497	156.522	0.010
5	3.830	1505.938	61	276.924	4.708	140	156.522	0.010	239	144.000	0.010	501	163.637	0.010
6	1.297	255.006	62	2.581	0.044	141	257.143	0.010	242	240.000	0.010	502	38.710	0.010
7	0.203	12.464	65	13.044	24.783	143	720.000	0.010	243	600.000	49.800	504	109.091	0.010
8	4.640	41.442	66	400.000	0.010	146	300.000	0.010	244	24.000	0.010	509	12.766	0.010
9	0.500	19.587	68	257.143	0.010	147	514.286	0.010	246	327.273	0.010	512	30.770	0.010
10	0.00673	0.291	69	83.721	3692.091	149	900.000	0.010	247	49.316	0.839			
11	0.053	0.351	70	400.000	233.200	150	600.000	0.010	248	514.286	0.010			
12	0.124	8.233	71	400.000	0.010	151	900.000	15.300	249	900.000	0.010			
13	0.998	1.231	72	300.000	0.010	153	50.000	0.010	250	450.000	0.010			
14	4.450	20.617	73	225.000	0.010	154	450.000	0.010	251	124.138	0.010			
15	3.258	1862.388	74	14.635	0.010	155	900.000	0.010	252	600.000	0.010			
16	1.685	7.889	76	900.000	3615.300	158	450.000	0.010	253	450.000	0.010			
17	2.162	42.703	78	400.000	0.010	160	3.013	3.013	254	9.091	14.091			
18	7.332	5.499	79	257.143	0.010	161	720.000	0.010	255	40.910	3.396			
19	5.547	35.040	80	150.000	4.950	164	97.298	0.010	256	12.245	0.010			
20	2.218	1.923	83	7.244	0.124	165	150.000	0.010	257	189.474	0.010			
21	31.579	8.937	84	300.000	0.010	170	720.000	0.010	259	900.000	0.010			
22	225.000	0.010	85	720.000	0.010	171	120.000	240.000	260	600.000	0.010			
23	62.069	1438.944	86	327.273	0.010	172	900.000	0.010	263	400.000	0.010			
24	1.905	830.666	87	600.000	0.010	176	600.000	0.010	264	900.000	0.010			
25	16.438	189.863	88	276.924	4.708	177	900.000	0.010	266	900.000	0.010			
26	211.765	0.010	89	189.474	3.222	181	450.000	0.010	267	92.308	1.570			
27	59.017	0.010	90	163.637	1270.964	182	900.000	0.010	268	900.000	0.010			
28	14.575	0.481	92	10.170	0.010	184	450.000	0.010	272	900.000	0.010			
29	6.040	0.405	95	600.000	0.010	185	450.000	0.010	273	720.000	0.010			
30	9.575	242.876	96	180.000	0.010	186	180.000	0.010	274	450.000	0.010			
31	0.806	14.278	97	150.000	2.550	187	514.286	0.010	275	400.000	0.010			
32	189.474	0.010	98	720.000	0.010	188	360.000	0.010	276	144.000	0.010			
33	360.000	0.010	99	189.474	997.958	192	720.000	0.010	279	211.765	0.010			
34	257.143	0.010	100	514.286	514.286	193	900.000	0.010	280	600.000	0.010			
35	11.356	283.344	101	21.687	0.369	199	300.000	0.010	281	24.490	0.010			
36	189.474	2542.168	104	360.000	0.010	200	300.000	0.010	285	51.429	0.875			
37	5.210	0.089	105	400.000	0.010	202	720.000	0.010	286	900.000	0.010			
38	2.824	0.048	106	300.000	5.100	204	600.000	0.010	287	900.000	0.010			
39	90.000	340.470	107	171.429	2.915	211	400.000	0.010	288	400.000	0.010			
40	156.522	988.748	108	180.000	909.000	212	240.000	0.010	293	720.000	23.760			
41	450.000	0.010	109	900.000	0.010	214	327.273	0.010	294	720.000	0.010			
42	124.138	0.010	110	13.900	0.010	216	720.000	0.010	301	720.000	0.010			
43	163.636	0.010	111	450.000	0.010	217	189.474	0.010	304	514.286	0.010			
45	720.000	2543.760	112	900.000	0.010	218	360.000	0.010	314	24.000	0.010			
46	138.462	0.010	115	600.000	10.200	219	200.000	0.010	315	720.000	0.010			
47	6.154	0.010	117	514.286	1996.972	220	40.910	3.396	338	720.000	0.010			
48	171.429	0.010	118	360.000	0.010	221	78.261	0.010	349	600.000	0.010			
49	171.429	514.286	119	34.952	0.010	222	13.187	0.010	383	600.000	0.010			
50	200.000	60.000	120	900.000	0.010	223	514.286	0.010	412	514.286	0.010			
51	128.572	0.010	121	600.000	0.010	225	600.000	0.010	416	43.374	0.010			
52	150.000	0.010	126	276.924	1998.554	227	600.000	0.010	431	65.455	0.010			
53	450.000	0.010	128	12.414	0.212	228	900.000	0.010	435	156.522	0.010			
54	514.286	0.010	129	450.000	0.010	230	900.000	0.010	438	600.000	0.010			
55	156.522	0.010	130	51.429	0.010	231	80.000	1.360	442	109.091	0.010			

Cited references

1. G. O. Collins, "Experience in automatic storage allocation," *Communications of the ACM* **4,** No. 10, 436–440 (October 1961).
2. J. K. Iliffe and J. G. Jodeit, "A dynamic storage allocation scheme," *Computer Journal* **5,** 200–209 (1962).
3. K. C. Knowlton, "A fast storage allocator," *Communications of the ACM* **8,** No. 10, 623–625 (October 1965).
4. K. C. Knowlton, "A programmer's description of L6," *Communications of the ACM* **9,** No. 8, 616–625 (August 1966).
5. H. M. Markowitz, B. Hausner, and H. W. Karr, *SIMSCRIPT—A Simulation Programming Language,* Prentice-Hall, Inc., Englewood Cliffs, NJ (1963).
6. D. T. Ross, "The AED free storage package," *Communications of the ACM* **10,** No. 8, 481–492 (August 1967).
7. B. Randell, "A note on storage fragmentation and program segmentation," *Communications of the ACM* **12,** No. 7, 365–372 (1969).
8. D. E. Knuth, *The Art of Computer Programming, Volume I: Fundamental Algorithms,* Addison-Wesley Publishing Co., Reading, MA (1968), pp. 435–455.
9. J. A. Campbell, "A note on optimal-fit method . . . ," *Computer Journal* **14,** No. 1, 7–9 (January 1971).
10. P. W. Purdom, S. M. Stigler, and Tat-ong Cheam, "Statistical investigation of three storage allocation algorithms," *BIT* **11,** 187–195 (1971).
11. J. M. Robson, "An estimate of the store size necessary for dynamic storage allocation," *Journal of the ACM* **18,** No. 3, 416–423 (July 1971).
12. B. H. Margolin, R. P. Parmelee, and M. Schatzoff, "Analysis of free-storage algorithms," *IBM Systems Journal* **10,** No. 4, 283–304 (1971).
13. D. S. Hirschberg, "A class of dynamic memory allocation algorithms," *Communications of the ACM* **16,** No. 10, 615–618 (October 1973).
14. J. S. Fenton and D. W. Payne, "Dynamic storage allocation of arbitrary sized segments," *Proceedings of IFIP 74,* North-Holland Publishing Co., Amsterdam (1974).

15. K. K. Shen and J. L. Peterson, "A weighted buddy method for dynamic storage allocation," *Communications of the ACM* **17,** No. 10, 558–562 (October 1974).
16. D. L. Russell, "Internal fragmentation in a class of buddy systems," Technical Note 54, Digital Systems Lab, Stanford University (January 1975).
17. J. E. Shore, "On the external storage-fragmentation produced by first-fit and best-fit allocation strategies," *Communications of the ACM* **18,** No. 8, 433–440 (August 1975).
18. C. Bays, "A comparison of next-fit, first-fit, and best-fit," *Communications of the ACM* **20,** No. 3, 191–192 (March 1977).
19. B. Cranston and R. Thomas, "A simplified recombination scheme for the Fibonacci buddy system," *Communications of the ACM* **18,** No. 6, 331–332 (June 1975).
20. H. R. P. Ferguson, "On a generalization of the Fibonacci numbers useful in memory allocation schema . . . ," *The Fibonacci Quarterly,* 233–243 (October 1976).
21. J. L. Peterson and T. A. Norman, "Buddy systems," *Communications of the ACM* **20,** No. 6, 421–431 (June 1977).
22. *IBM Virtual Machine Facility/370 Introduction,* GC20-1800, IBM Corporation; available through IBM branch offices.
23. J. P. Buzen and P. J. Denning, "Measuring and calculating queue length distributions," *Computer* **13,** No. 4, 33–44 (April 1980).
24. J. D. C. Little, "A proof of the queueing formula $L = \lambda W$," *Operations Research* **9,** 383–387 (1961).
25. P. F. Wyman, "Improved event-scanning mechanisms for discrete event simulations," *Communications of the ACM* **18,** No. 6, 350–353 (June 1975).
26. C. J. Stephenson, "Fast fits—New methods for dynamic storage allocation," to be published in *ACM Transactions on Computer Systems.*

General references

M. J. Bailey, M. P. Barnett, and P. B. Burleson, "Symbol manipulation in FORTRAN-SASP I subroutines," *Communications of the ACM* **7,** No. 6, 339–346 (June 1964).

A. T. Berztiss, "A note on the storage of strings," *Communications of the ACM* **8,** No. 8, 512–513 (August 1965).

CMS User's Guide, GC20-1819, IBM Corporation; available through IBM branch offices.

Gerald Bozman *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Mr. Bozman initially joined IBM in 1961 as a systems engineer in the White Plains branch office. After leaving IBM in 1965, he worked on the development of communication and time-sharing systems. He returned to IBM in 1977 as a systems programmer with the former Office Products Division in Franklin Lakes, New Jersey. He is currently a member of the Computer Sciences Department at the Research Center. Mr. Bozman received a B.S. in English from Columbia University and an M.S. in computer and information science from the New Jersey Institute of Technology.

William Buco *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Mr. Buco is manager of the VM/370 Systems Programming project at the Research Center. From 1970 to 1974 he worked at the IBM Cambridge Scientific Center on prototype versions of Discontiguous Shared Segments and scheduler extensions for VM/370. From 1974 to 1977 he worked at the Research Center as a systems programmer improving the performance and reliability of VM/370. He has been in his present position since 1977. In 1974 Mr. Buco received a B.A. in mathematics from Northeastern University and in 1977 an M.A. in computer science from Columbia University.

Timothy P. Daly *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Mr. Daly worked for Unimation, Inc. from 1976 through 1978 as a computer specialist in the field of industrial robots. He joined IBM in 1978 to work on improving the performance of VM/370. Currently he is working at the Research Center enhancing the AML Robot Language. Mr. Daly received his B.S. in mathematics from Montclair State College and his M.S. in computer science from Fairleigh Dickinson University.

William H. Tetzlaff *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Mr. Tetzlaff joined the Service Bureau Corporation in 1966. He joined the Research Division of IBM in 1969 and has done research in the areas of information retrieval and system performance. He published several papers on that research, and received an IBM Outstanding Contribution Award for his work on system performance. He recently completed a temporary assignment as a member of the Technical Planning Staff of the Research Division. Mr. Tetzlaff studied engineering sciences at Northwestern University and is a graduate of the IBM Systems Research Institute. He is currently manager of VM Analysis and Restructure in the Computer Sciences Department of the Research Division.

Reprint Order No. G321-5209.