Storage management in IBM APL systems

by R. Trimble

APL systems have traditionally used specialized storage management schemes that avoid storage fragmentation by "garbage collection," moving live data as needed to collect unused storage into a single area. This was very effective on systems with a small amount of real storage addressed directly. It has become less effective on today's systems with virtual addressing and large amounts of virtual storage. Both traditional schemes of storage management and a recently implemented replacement for them are described. The focus is on implementations for IBM mainframe hardware.

Programs written in compiled languages typically use static definitions of working storage. Much of the time the language syntax requires that variables be declared as a particular type, structure, and often a particular size. This allows the compiler to generate very specific code for accessing the variables.

In contrast, interpretive programs typically provide much less data declaration information, and declarations are often implicit in the data usage. A number of interpretive languages allow a single variable to take on varying definitions at different times during program execution. APL, in fact, has no data declaration constructs at all for objects that exist within the active workspace. An object may change during execution from Boolean to real to complex, from simple scalar to four-dimensional array to nested structure, and from numeric to character to defined function.

Depending on the point of view, this dynamic characteristic of data has been described as introducing anarchy into the language, forcing heavy execution time overhead, or permitting powerful and elegant algorithms that are independent of data structure and format. Less frequently analyzed is the impact on storage management strategies, and the secondary impact of those strategies on total system performance. This paper discusses the storage management schemes that are used for APL running on IBM mainframe processors.

APL data organization

By necessity, APL objects must be completely self-describing, and it is impractical to assign them fixed locations or sizes. This leads immediately to a level of indirection in locating named objects accessed by programs or users. Ultimately the locator technique must provide for a symbol table lookup, since new references to objects can be introduced interactively at any time. In practice, though, a symbol table search incurs too much overhead on every reference, so a pointer table with statically assigned slots is used, each slot pointing to the current location of the associated data. Programs needing to access a data object can retain a table index for that object instead of its actual address.

Traditional APL implementations are contrasted here with systems like LISP that have large numbers of internal connections among relatively small stored objects. Some APL systems, such as VS APL,[1] did use internal synonym chains to avoid making copies of objects, but in general APL systems have

handled large array objects that were mostly *externalized*, or named. LISP and some other languages typically use direct internal pointers from one object to another, and this is the only reasonable approach when storage cell sizes are very small. A full pointer table for LISP could easily use up a quarter or more of all available space in the system, and management of space within it could become a severe problem.

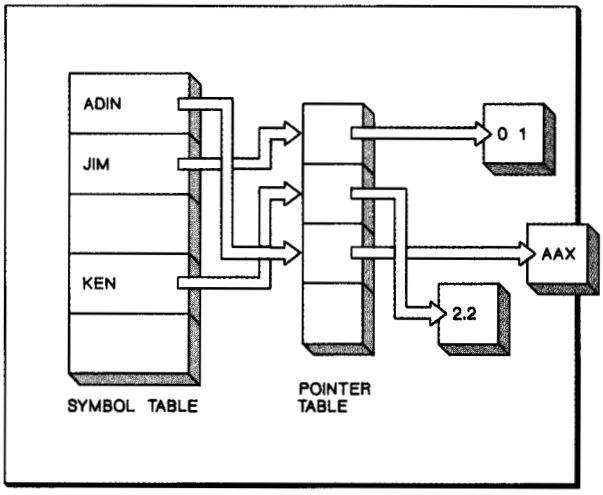
APL2 has introduced nested arrays into the language, and this has significantly increased the number of internal connections, but the array orientation remains. For this reason, and to avoid the decision overhead of handling a mixture of direct and indirect pointers, APL2 follows traditional APL usage of making all pointers indirect.

There are two major ways in which pointer tables have been implemented by APL systems. Figure 1 shows separate symbol tables and pointer tables. This approach permits the symbol table to be structured for binary or tree searches, and to be reorganized or expanded as needed.

Figure 2 shows a combined symbol and pointer table. The names of objects are stored as if they were objects themselves (though some systems store short names directly within the table). To locate a symbol by name, the system must follow the name pointer from each row of the symbol table. The combined table requires less storage, but is not amenable to table reorganization, since an unknown number of indices into it exist throughout storage. Typically a hashing scheme is used to locate names within the table, but this precludes dynamic expansion of the table. Table expansion would be possible only if sequential searches were done (which are very costly in time) or if an index were maintained (still significantly more costly than hashing). For these reasons, systems employing the combined table normally have a fixed symbol table size, or a size that can only be set when an APL workspace is first created.

The combined table was used in earlier APL systems, including IBM's APL\360, APLSV, and VS APL. APL2, IBM's current offering for the IBM System/370* and System/390*, uses separate symbol and pointer tables, in large part because of nested array extensions, but also partly because it was designed for large applications with more objects, making symbol table expansion much more important.

Figure 1 Separate tables for locating objects

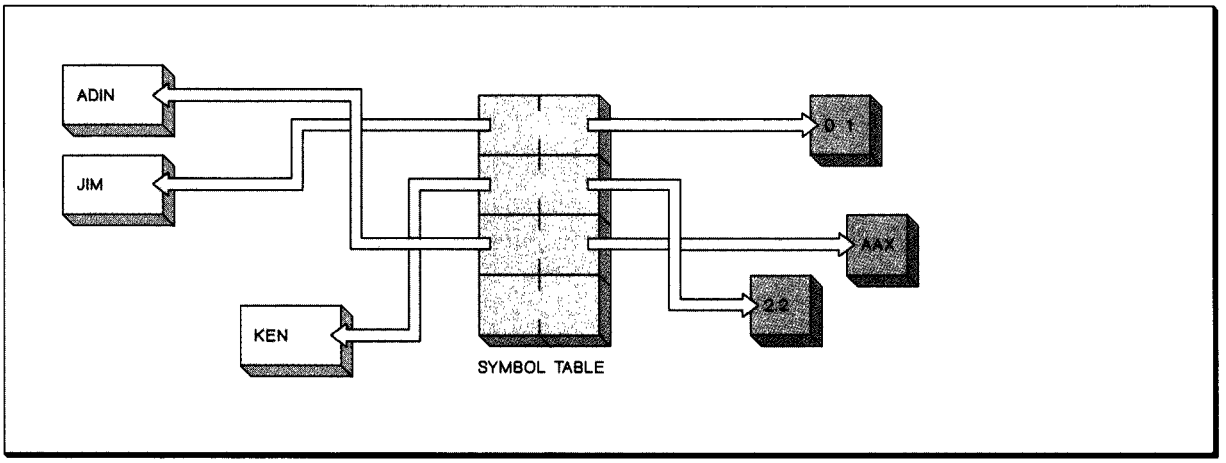


Whichever structure is used by an implementation, the primary pointer table contains addresses of all other objects in the APL workspace, and that is the only place (with occasional exceptions) such addresses are kept. Since interpreter routines always maintain a direct pointer to the pointer table, there is very little extra cost in converting a table index to the address of the corresponding object. Most importantly for storage management, it is also easy to move an object from one place to another, since only a single pointer to it needs to be updated. One other rule is enforced to make this possible—any pointers to locations within an object are always stored as offsets, not addresses.

Traditional APL storage management

Another attribute of APL objects is that many of them are very transient. APL programs often use simple names like *X* for variables that contain many different kinds of data during the execution of a single defined function. Since the storage requirements for these various usages may gyrate wildly, the system actually creates a new object each time a value is assigned to the variable, and discards the object that previously represented the variable (thus their transient nature). Also, because APL is an array processing language, intermediate results are arbitrarily large and it is not practical, in general, to use predefined temporary areas to hold those results. Thus each processing step within an APL statement produces a new object as its result.

Figure 2 Combined table for locating objects



Because of these characteristics, storage allocation and release are critical paths in the performance of APL systems. Operating system path lengths for allocating and freeing storage are typically hundreds or thousands of instructions. If APL were to use those services for each object allocation, they could easily use up 90 percent or more of the application execution time. Thus APL, along with a number of other languages, was compelled to provide its own storage management function within an area (which APL calls an active workspace), obtained from the operating system.

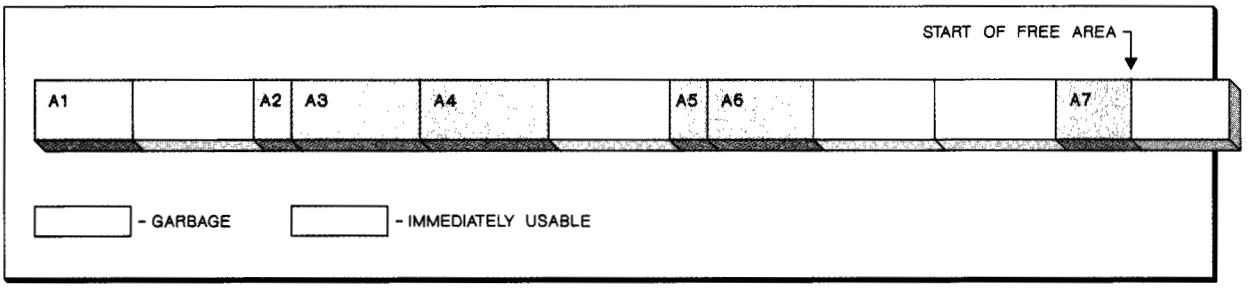
The traditional APL storage management technique is very simple, but extremely fast in processing time. The APL system adds a standard prefix to all objects. The size and format of the prefix have varied among APL implementations, but the prefix has included at least a flag (typically the first bit) that indicates whether the area is currently in use or is *garbage*, i.e., data no longer needed, and a field containing the length of the area.

A pointer is maintained to the beginning of a free area where it is known that no storage is currently allocated. When an allocation request is made, the storage is allocated at the beginning of the free area, and the free pointer is stepped beyond the new allocation. When an area is freed, its *garbage* flag is set. (Often the end of the freed area is checked against the free pointer; if they match, the free pointer is backed up, but this is not a necessary part of the algorithm.) Figure 3 shows a simple example of what a workspace might look like after a few such storage operations.

Eventually, of course, the free pointer will approach the end of the free area, and a storage request will be made that cannot be satisfied. This triggers *garbage collection*, which has a number of meanings in computing literature:

1. Garbage collection sometimes refers to the process of determining which parts of storage can be reused, perhaps by following all valid storage links. APL, as was already indicated, maintains a garbage bit in each block of storage. It also maintains (either in the storage block or the pointer table) a *use count* (i.e., storage is in use) field for each active block. The garbage bit is turned on when the use count goes to zero, so there is no ambiguity about which blocks of storage can be reused.
2. When a garbage bit is available, the system normally, at some time, scans storage looking for blocks it can reclaim. Often part of that scan involves coalescing adjacent garbage blocks. APL garbage collection performs this process.
3. Blocks identified as containing garbage may be chained together for later reuse. This has not typically been done by APL systems, because it does nothing to relieve fragmentation and es-

Figure 3 Sample of workspace with garbage, where A1-A7 represent allocation requests that have been satisfied



sentially leads to the same sorts of operating system storage management schemes and path lengths that APL has tried to avoid.

4. "Live" blocks (those containing data that are currently in use) may be moved, resulting in adjacent areas of garbage that may be collected into larger garbage blocks. Since APL maintains a complete indirect pointer list, it is relatively simple to move live entries. (APL systems normally maintain a pointer list index in the live entries, which makes it trivial to locate the one pointer which must be updated.) So for APL, garbage collection is the process of returning all of the garbage areas to the block of free storage.
5. There are several possible algorithms. APL systems have almost universally used a "shifting" rule that keeps the live storage blocks in their previous order. The advantage of this is that over time the more static objects in the workspace will migrate to the low-address end, and will be unaffected by later garbage collections. (Typically a "lowest garbage" pointer is maintained so that the system can skip over the static part of the workspace.) The disadvantage is that in the short term very large amounts of storage may need to be moved to make small but previously long-lived blocks reusable.

Some APL systems have used *predictive garbage collection* techniques that do the storage compaction as soon as a certain amount of garbage has accumulated. This approach can eliminate long pauses for garbage collection at unexpected times, but typically also increases the number of times that an object will be moved before it reaches its final resting point (or is deleted). Thus the net effect of such schemes is to increase the total amount of storage movement in the system, and so increase the CPU time used in processing an application. The approach can be useful despite this characteristic, both because it does yield a more predictable response time, and because it can reduce the application working set. More will be said later about that aspect.

One other enhancement used by VS APL[1] was to "ping-pong" allocations between both ends of the free area. It did this by maintaining floating pointers to both the beginning and the end of the free area, and by alternating their usage. The usefulness of this becomes apparent when we consider what happens while processing a series of primitive functions in an APL statement. For example:

```
A←1 3 2.5
A←2+3×⌊A
```

Figure 4 illustrates the sequence of allocations with a normal single-ended workspace system. Note that each primitive operation must obtain space for its result and calculate it before the space for the previous temporary result can be released.

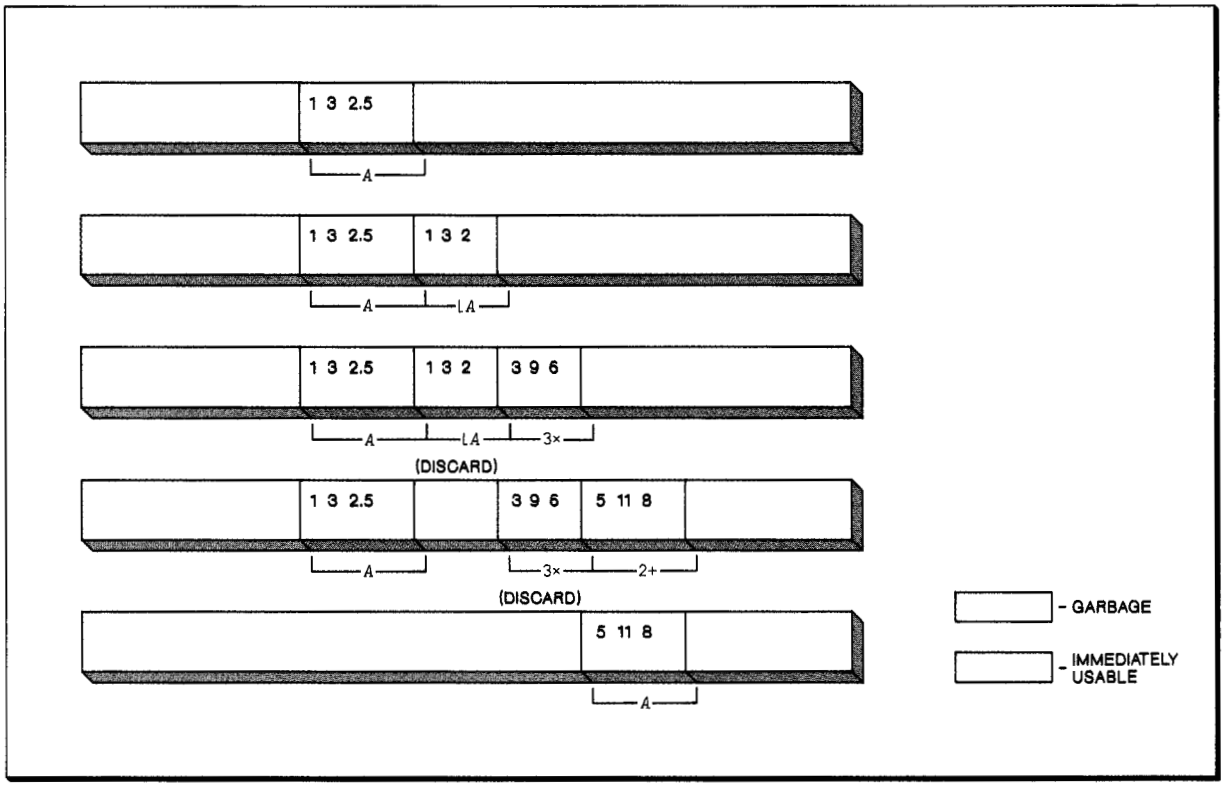
Figure 5 shows the corresponding sequence of allocations with a two-ended (ping-pong) workspace.

Because of the ping-ponged allocations, temporary blocks can often be returned immediately to the free area, and embedded garbage builds up more slowly.

The costs of garbage collection

The first APL implementations ran on systems without paging facilities and used 32K-byte work-

Figure 4 Processing with a one-ended workspace



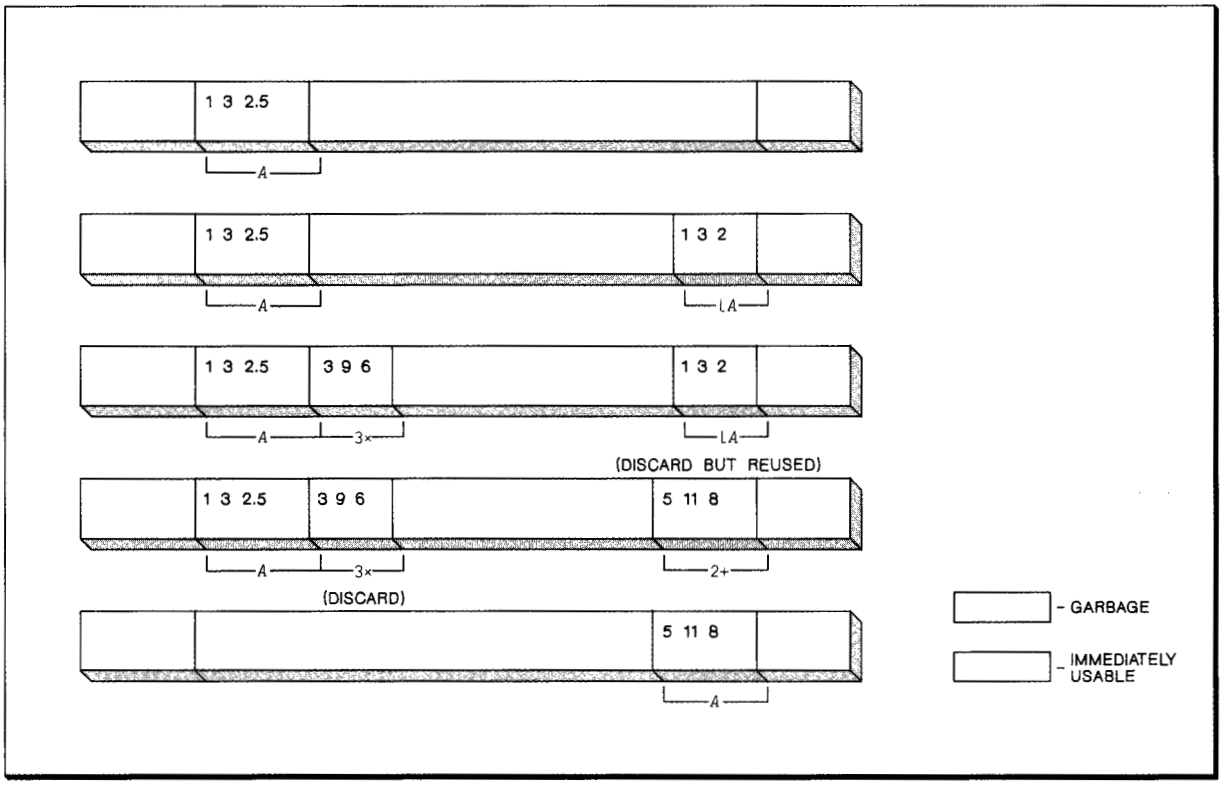
spaces. A typical garbage collection would move 4K bytes of data or less, and might use the time equivalent of less than 1000 instructions.

Today's APL products run on systems that are often capable of supporting workspaces up to a gigabyte or two in size, all in pageable virtual storage. Although many users limit themselves (or are limited by their installations) to 10-megabyte workspaces or less, a significant number are routinely using 50–100 megabytes or more. In typical cases only a small part of these larger workspaces is used for static data and functions. The extra space has made it possible to manipulate multiple megabyte arrays and use algorithms with very large intermediate results. But this in turn means that garbage collections often involve moving many megabytes of data.

A typical garbage collection for a 20-megabyte workspace might move 2–4 megabytes of data, requiring execution time equivalent to executing on the order of 100 000 instructions. But this is only the beginning of the problem. In the process of locating and moving the data, the APL system will probably touch 75 to 80 percent of the pages in the workspace, or around 400 pages for the 20-megabyte example. On typically loaded multiuser systems a significant number of these will be paged out, resulting in long delays to retrieve them, one after another. These delays can easily add up to execution pauses of 5 to 10 seconds, which is intolerable in an interactive system. These sudden paging loads can also trigger periods of saturation for the paging devices, and thus lead to execution pauses for other interactive users on the system.

Finally, periodic usage spikes of real storage caused by garbage collection mislead system resource management programs, causing them to overestimate future APL real storage requirements and frequently to move APL users to a lower priority service class for most of their processing.

Figure 5 Processing with a two-ended workspace



The scenario just described at 20 megabytes becomes much (more than ten times) worse at 200 megabytes. The system has a limited amount of real storage, and only a fraction of that can be dedicated to a single user. It is a rare system today that will allow one user to control as much as 150 megabytes of real storage at a time. Note that APL garbage collection actually involves two pointers "floating up" through the workspace, one for where blocks are being moved to and the other (many megabytes ahead of the first in such a huge workspace) for where blocks are being moved from. When the distance between those pointers exceeds about half the real storage of the available user storage in the system, the pages will begin to be paged out and back in again between the time they are used. This can triple or quadruple the paging load described above. Execution pauses of many minutes have been reported under these circumstances.

As was indicated earlier, predictive garbage collection, taking action when a threshold is reached on uncollected garbage, actually increases the total amount of storage movement and thus the processor time required to run a given application. Despite this, it can be useful, because the paging usage spikes are significantly reduced, and the "moved to" and "moved from" pointers are much closer together during a garbage collection.

Using quickcells to minimize garbage collection

APL2 uses one very successful strategy to reduce the number of garbage collections. Although object allocations come in many and varying sizes, it was noted that a large number of them are quite small. This is particularly true for APL2, which actually uses two or more separate storage areas for most nonscalar data objects. One of the areas contains the data objects themselves and the other contains the description of the data. (Nested arrays include a number of descriptor areas and data areas.)

- The standard descriptor blocks for vectors, matrices, and three-dimensional arrays can all be fit into a 32-byte area.
- The same size area can hold the data for an array of 1–3 real numbers, up to six signed integers or 24 characters, and as many as 192 Boolean values.

APL2 storage management includes special handling for 32-byte storage cells. When such a cell or *quickcell* is no longer needed, it is put on a special chain instead of being marked as garbage. Then when another area of that size is needed, the chain is checked first, thus avoiding encroachment into the free area in many common situations. Since the chain is maintained in last-in-first-out order, the cell selected from the chain has the highest probability of still being in real storage.

There is another performance advantage to these quickcells. Any time a new storage area is created from what was the free area, a free slot in the pointer (or symbol) table must be located and associated with the area, and the required header area must be formatted. The quickcells retain their table slot and are already formatted. Using them, APL2 was able to achieve a path length of about 30 instructions to allocate a 32-byte area, including call and return overhead, necessary tests, and additional formatting specific to the type of area being obtained.

APL2 also provides a separate quickcell pool for each type of scalar data (a single unstructured character or number). There are six of these pools, covering everything from standard characters to complex numbers. Four of those six (characters, extended characters, short integers, and signed integers) need only a 16-byte area, so the system splits a quickcell to form two "short scalars." Allocation path lengths for all of the scalar quickcells are a trivial 11 instructions because special entry points are used, only one test (for empty pool) is needed, and the cell is already completely formatted except for the actual value.

Of course it is possible for APL applications to use a very large number of such areas for a short time, leaving huge pools of quickcells behind. The normal garbage collection algorithm would not detect those, but APL2 provides a special quickcell cleanup routine that does release the table slots and mark the cells as garbage. This is usually performed if a standard garbage collection is not able to free up enough space. Until such time as this happens, though, large quickcell pools can have the effect of increasing the number of real pages required by the application.

Using buddy-system cells to avoid garbage collection

It is tempting to try to extend the advantages of quickcells to larger areas. This must be done carefully, though, because as the number of special classes is increased the cost of determining the appropriate class can rise, and pools of storage can grow in some classes at the expense of available space for others. One interesting solution to this dilemma is a storage management technique called the "buddy system."

Knuth[2] reports that H. Markowitz first used the buddy system for SIMSCRIPT, but it was apparently first published by Knowlton[3] and may have been named by Knuth. This early work has now come to be called a "binary buddy system." Hirschberg[4] proposed a more space-efficient buddy system based on a Fibonacci series, and Cranston and Thomas[5] described a simplified recombination scheme for Hirschberg's system. Shen and Peterson[6] took a different space-saving approach, which they called a "weighted buddy system." Then Peterson and Norman[7] produced a paper that reviewed the various buddy schemes and concluded that either the original binary system or the improved Fibonacci system was preferable. Bozman et al.[8] found buddy systems in general very fast but inferior to subpool-based systems for their purposes with IBM's VM/SP product. This may have been because the binary system described below required an additional double word in each allocation. As will be shown, no extra storage is needed for APL.

Unfortunately for APL2, the improved Fibonacci system depends on availability of three status bits in the storage block, which would require a major restructuring and reinterpretation of flags throughout the interpreter. So the following information focuses on the original binary system. It should be noted that Page and Hagins[9] have more recently defined an improved weighted buddy system, but we have not analyzed that for applicability to APL.

The binary buddy system works by allocating all storage in sizes which are a power of two. Free area chains are maintained for each storage size. If no storage is available on a particular chain, an area can be taken from the next larger size and split to form

two areas, one of which will be put on the chain and the other used to satisfy the current request. (This splitting is, of course, a recursive process since the next larger chain could also be empty.)

Consider what would happen if a request were made for 80 bytes of storage and the system currently had no free blocks smaller than 4K bytes. The initial request would be rounded up to the next power of 2 (128 in this case) and then recursive splitting would be used to satisfy it. An implementation can choose which of the two "buddies" created by splitting an area is to be used immediately, and which is to be placed on the chain. For this example we assume that the buddy at the lower address is placed on the chain. Figure 6 is a pictorial representation of the way the 4K-byte storage area would be divided up at the end of the request.

First, note that any request for a small amount of storage when pools are empty will not only get that storage but will prime all pools up to the next one that was not empty. Because of this behavior, pools tend to be nonempty much of the time, and a majority of storage requests can be satisfied without having to split larger cells.

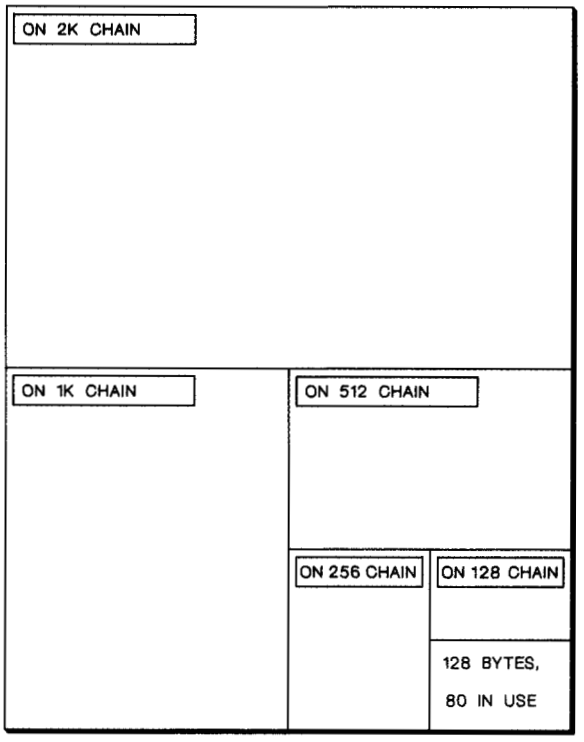
A second less obvious observation is critical to the behavior of the buddy system when storage is returned. If the original 4K area illustrated in Figure 6 began on a 4K-byte boundary, then the 2K buddies will each be on a 2K-byte boundary, 1Ks on a 1K boundary, and so forth, no matter how many times the area is split. In general any buddy system cell must be on a boundary that is a multiple of its size, and that requirement will be met automatically so long as it was met by the original areas. A different way of stating this is that for any buddy cell of size 2^n, the low order n bits of its binary address will be zero.

Now consider what happens when a cell of size 2^{n+1} is split. The first (low address) cell created will have $n + 1$ low order zero bits in its address, while its buddy will have the same address except for a 1 in the first of those $n + 1$ low order bits. Splitting a 1K (2^{9+1}) cell, for example, yields Boolean addresses of

```
xxxx xxxx xxxx xxxx xxxx xx00 0000 0000
xxxx xxxx xxxx xxxx xxxx xx10 0000 0000
```

But that same bit position is the location of the sole 1 in the binary representation of the length (2^n) of the new buddy cells. This leads to the remarkably

Figure 6 Dividing a 4K storage area to satisfy an 80-byte request



useful conclusion that given any buddy cell, performing an *exclusive-OR* operation of its address with its length will yield the address of its buddy.

The exclusive-OR technique makes it feasible to coalesce cells without an unreasonable amount of processing if two pieces of information are available with each cell:

- A flag that indicates whether the area is currently in use
- A field containing the length of the area

These are the same pieces of information that APL storage systems have always maintained. The buddy can be located by using the exclusive-OR operation. Once located, the two areas can be coalesced if the buddy is not in use and if it has not been further subdivided, i.e., if its length has not been reduced.

Knowlton's original paper expressed one concern that most later researchers seem to have ignored.

He felt that it might be better not to coalesce buddies in all cases where that was possible. Based on some modeling we indeed found what we called a "zipper" effect that can occur if a single small storage area is repeatedly allocated and freed at a time when most chains are empty. (Getting the block causes a series of cell divisions, leaving one cell on each chain. Freeing that block then zips all the cells back together into one large area, leaving the chains empty again.) Kaufman[10] has looked at this in some detail and considered two types of solutions:

1. Leave a minimum number of cells on each chain, with the number probably determined by the usage level of the chain
2. Delay recombination until a larger cell is needed

He concluded that there were conditions where each solution would be better than the other. For our work we chose a simplified form of the first solution, bypassing the coalesce if it would leave the chain empty.

It would not be fair to leave this topic without acknowledging one significant problem. Traditional APL storage allocation techniques rounded storage sizes up to a multiple of eight, while buddy (plus quickcell) allocation rounds sizes to a power of two with a minimum of 16 bytes. This has been referred to in the literature as *internal fragmentation*, and can result in an effective virtual storage utilization of only about 75 percent.

That number can be intuitively understood by observing that all of the storage areas allocated to a given buddy cell size are at least 50 percent as large as that cell, and at most 100 percent of the cell size. Assuming a linear distribution of sizes, the size of the required storage would average 75 percent of the cell size. In practice, size distributions are skewed with more allocations of the smaller sizes, so that the typical utilization should be somewhat less than 75 percent. Compensating for this is the fact that more than half of the allocations are for either scalar quickcells or array descriptors in quickcells. And it happens that those always use at least 75 percent of the cell size.

Buddy system researchers have also explored *external fragmentation*, which occurs because multiple unpaired but unused cells of some size may exist and yet be unusable if a larger block of storage is needed. This fragmentation is not a conceptual problem for APL, because active cells can be swapped at any time so that the unused cells do become buddies. It can, however, have some practical effect, because the swapping process can be time consuming for the CPU, and can increase the real storage requirements of the system.

Managing large-scale accountable storage

It is not very practical to extend buddy cell sizes beyond 4K bytes on an IBM System/370, because in most cases operating system interfaces do not provide for storage alignment on any boundary greater than 4K. But this is not a serious problem for two reasons:

1. The number and frequency of large allocations is far lower than for small allocations.
2. Once a large area has been allocated, a great deal of effort normally goes into filling it with data.

Both of these reasons ensure that path lengths for large area allocations are not critical. Any of a number of more conventional storage schemes could be used successfully to provide accountability and reuse of large areas. Indeed, it would be feasible to depend on operating system storage management for these areas. APL systems do not do that at the present time because of a concern about storage fragmentation. If storage should become badly fragmented there would be no practical way to recover when using operating system control. So long as APL controls the storage, garbage collection can be used if necessary.

Along with the work to support buddy system cells, there is also a new scheme for managing larger blocks of storage. Historically such schemes have usually been based on maintaining linked lists of available areas. (Each currently unused area contains a pointer to the next unused area in its group.) Since we were dealing specifically with large amounts of pageable storage we were concerned about the potential paging overhead of traversing such chains to locate a usable area.

The solution chosen was to maintain a bit map of storage blocks. This became feasible because the smallest unit of storage to be managed was a 4K page (2^{12} bytes). All subdivisions of that were managed by the buddy system. Because of operating system limitations, the largest total area that APL could be presented with was somewhat less than 1 gigabyte (2^{30} bytes). This meant that all possible pages could be represented by $2^{30-12} = 2^{18}$ bits. This

is 2^{15} bytes, given an 8-bit byte. Thus a bit map for the largest supported amount of storage could be stored in 32 K, or eight 4K pages, a very reasonable amount of space when dealing with a gigabyte of storage. For workspaces up to 128 megabytes the bit map requires only a single page.

One of the problems that linked list management systems must address is coalescing adjacent free areas. This problem disappears with a bit map, since the bits are stored in virtual storage order. Linked list systems can also simplify the problem by using address order for linking, but this usually makes allocation and de-allocation searches too expensive. With a bit map there is no problem at de-allocation time. The storage address is trivially converted into an index into the table. But locating an available area of appropriate size during allocation is another matter.

Bit maps are used for pages and buddy system cells are used for smaller cells.

This was solved by using a set of 256-byte lookup tables to convert one 8-bit pattern to another. A table is chosen based on the number of pages required for an allocation. Each byte in the bit map is treated as an index into the table. The content of the table entry indicates whether the request can be satisfied from that section of the bit map.

If, for example, a request was made for six pages of storage, the request could be satisfied by either

- Six or more contiguous free bits within a byte
- Three or more contiguous free bits at the edge of a byte with the remaining one to three bits available at the adjacent edge of the adjoining byte

The bit configurations satisfying the first criterion are:

```
00000011 00000001 10000001 11000000
00000010 00000000 10000000 01000000
```

Treating these bit configurations as binary numbers, zero-origin entries 3, 1, 129, and 192, as well as 2, 0, 128, and 64 in the lookup table would need to contain values that indicate the configurations are satisfactory.

The System/370 includes a translate and test (TRT) instruction that can automate the search, so long as unsatisfactory configurations have a binary zero entry in the lookup table. Because of this a convention of putting the one-origin offset of the first satisfying bit into the table was chosen.

Seven tables of this form were generated, to allow searches for up to seven contiguous available bits in a byte. Note that if the search succeeds, both the byte and bit numbers of the desired position in the bit map are known.

When fewer than eight pages are required, a fast search is made for all bits within one byte using one of the above seven tables. If this search fails, a byte-by-byte check of the bit map is used to look for an area crossing two bytes. This check also utilizes a lookup table that is indexed by the bit map bytes, but in this case the indexing is done manually, and the codes within the table indicate the number of bits available on each edge of the argument byte (i.e., the code is treated as a pair of 4-bit numbers). By adding appropriate edge-counts from adjacent bytes, the system can determine whether enough space is available at that boundary.

If eight or more pages are needed, a search is made for a byte in which all eight bits are free. Once such a byte is found, the search is expanded around that byte as needed to obtain more than eight pages. If no appropriate area can be found containing an all-free byte, and if 14 or fewer pages are needed, the same edge search is run that is used for less than eight pages.

The expanded search for more than eight bits is somewhat tedious, but it should be noted that the storage areas involved are always longer than 32K bytes, so the processing cost after allocation is usually much larger than the time spent to locate an available area.

In all of the searches a choice had to be made between a "first fit" (or perhaps "next fit") and a "best fit" rule. Bays's analysis[11] shows that next fit is a poor choice, but does not provide a clear preference between first and best fit. We prototyped an

exact fit scan followed by a first fit scan, but found that for our bit map search routines using first fit alone provided slightly better overall performance. We did not analyze the reasons, but assume it was a combination of the extra CPU cost for a double scan and because first fit develops a set of "favorite pages," or those which are less likely to be paged out.

As with the buddy system, there is a storage penalty for the page-oriented allocations. For blocks up to 8192 bytes (8K) the same usage constraints exist as for buddy cells, and the effective utilization is slightly less than 75 percent. This number rises, though, for larger blocks. The only allocations made to a ten page block, for example, are those requiring more than 90 percent of its space. The usage of large arrays varies greatly among applications, so it is difficult to generalize. It is probably safe to say, though, that for most applications the effective utilization of large-scale storage will be between 75 and 95 percent.

Getting the best of both

The previous discussions of buddy cells and large-scale storage each ended with warnings about limits on effective utilization of virtual storage. To some extent this is a deceptive concern. All storage management systems produce small fragments of storage intermixed with live data, and for most systems the fragments are either completely unusable, usable only at great expense, or usable only for a small subset of the allocation requests. But there are three ways in which this is a very real concern:

- The fragments at issue are all less than one page long, and are all on pages containing live data. Thus in a paging system they always act to increase the number of real pages required to run the application effectively.
- Unlike traditional APL storage management, there is no way to "squeeze" the fragments out of the live data and make them available again.
- Because of the previous point, the unusable fragments would still exist in APL workspaces that are saved. This implies an increase in required permanent storage space as well as additional data transfer while reading and writing the workspaces.

To address these concerns a hybrid scheme was implemented. The workspace is divided into two sections, with a floating boundary between them. Storage to the left (low address end) of the boundary contains densely packed objects managed using traditional garbage techniques. Storage to the right of the boundary is managed using bit maps for pages and the buddy system for smaller cells. So long as enough reusable storage is available at the right end of the workspace, garbage is allowed to collect at the left end. In many cases this will suffice for so long as the workspace is active. When a request arrives that cannot be satisfied, some form of garbage collection is done. One of three alternatives is chosen:

1. If there are enough free pages at the right end to satisfy the request (but they are scattered), and there is more storage available in free pages than in garbage at the left end, then allocated pages at the right end are rearranged so that all free pages are in one group.
2. If there is enough garbage at the left end to satisfy the request, and there is more storage available in garbage than in free pages at the right end, then all garbage at the left end is collected, the dividing line is moved left to the end of the last page on that side still containing data, and the remainder of the collected storage is made available in the page pool.
3. If neither end has enough space to satisfy the request on its own, all unused quickcells are released and then a full garbage collection of the workspace is done. At the end of this process the dividing line is at the end of the last page containing data, and the remainder of the workspace is in the page pool.

The third form of garbage collection is always performed when a workspace is saved. (The page pool is not kept with the saved workspace. Indeed when the workspace is reloaded later the page pool may be of a different size.)

Note that this concept of two storage zones is a simplified form of "generational garbage collection" as recently advocated by Appel,[12] Wilson and Moher,[13] and others.

Comparative performance measurements

A limited amount of performance measurement has been obtained comparing APL2 with and without the storage management changes described in this paper. The results are very encouraging, but should not be over-interpreted. A storage-intensive test function was generated that allocated and ini-

Figure 7 Comparison of CPU times

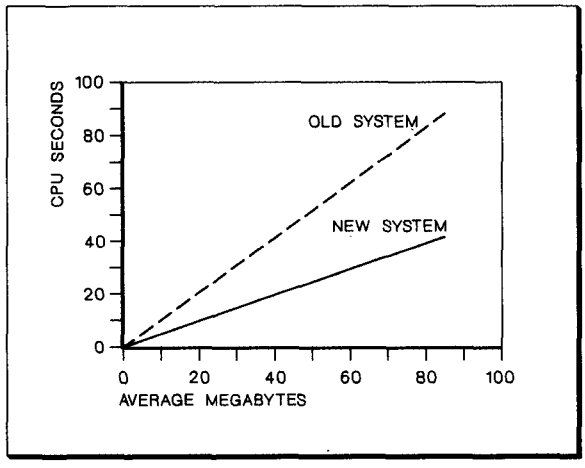
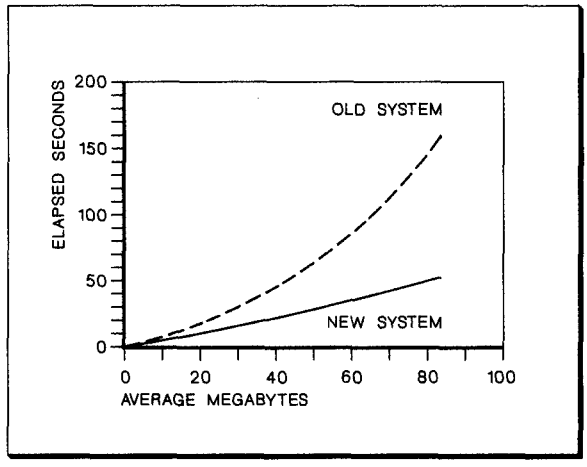


Figure 8 Comparison of elapsed times



tialized storage blocks of random sizes and random lifetimes. The algorithm automatically adjusted to workspace size, and tended to keep an average of 60 to 65 percent of the workspace in use. This is not a typical APL application, but it was created specifically to exaggerate any differences in the storage behavior of the systems.

Figure 7 shows the amount of CPU time used by the test case over a range of workspace sizes. It has been scaled by the average amount of allocated storage rather than by workspace size to remove any bias due to buddy cell internal fragmentation. All tests were run on an IBM 3090* with 128 megabytes of real storage and very little other concurrent activity, so no paging was needed.

Figure 8 shows test case elapsed times from the same runs as Figure 7. We believe that the accelerating slope seen here is caused by IBM's Multiple Virtual Storage Resource Manager function intentionally slowing the application down as larger fractions of the system's total real storage are used.

Testing under loaded conditions produces similar results. As a controlled environment, ten tests were submitted simultaneously and competed for three initiators on an idle system with three CPUs. Separate runs were made with 100-megabyte and 200-megabyte workspaces. With three initiators and three CPUs these resulted respectively in roughly 1.5:1 and 3:1 overcommitments of available real storage. At 100 megabytes the new system used 47 percent as much CPU time and only 38 percent as much elapsed time as the old. At 200 megabytes the elapsed time dropped to 34 percent. In both the 100- and 200-megabyte cases the usable allocated storage dropped by less than 1 percent since internal buddy cell fragmentation is of little consequence in such large workspaces.

Finally, it is important to stress again that the differences shown here are exaggerated from those that would be seen by an APL application. More than 90 percent of the test application time was spent in allocating and initializing storage. It would be more typical for an application to spend between 1 and 10 percent of its CPU time in that code, and it could spend much less than 1 percent of its elapsed time there if it was highly input/output oriented.

Concluding remarks

For 25 years APL systems have depended on garbage collection for storage management, and it has served them well. Pure garbage collection schemes are likely to be used less in the future than in the past, but composite schemes will continue to exist where garbage collection is an important component.

This paper has focused on the current storage management schemes for APL running on IBM mainframe hardware and their operating systems. The issues and solutions would be entirely different, for example, if the storage model used by an IBM Ap-

plication System/400* processor were assumed. This paper has not addressed the unique attributes of Enterprise Systems Architecture systems, but the virtual storage model that they implement is not radically different from their predecessors. It does hold out the promise of breaking the 1-2 gigabyte barrier that was assumed earlier in this paper. Unfortunately it appears the promise can be realized for APL2 only with a major rewrite of the interpreter, and that work has not been accomplished. It would be premature to speculate on optimal storage management strategies for multiple address spaces.

One clear lesson of the last four decades is that computer addressability will quickly expand beyond anything we consider reasonable today. The more sobering lesson is that application storage requirements seem quite capable of expanding as fast as hardware capabilities. This race will not only keep implementers of language products busy for the foreseeable future, it will also keep a noticeable part of their focus on matching these storage requirements and capabilities.

* Trademark or registered trademark of International Business Machines Corporation.

Acknowledgments

Brent Hawks and James Brown did an outstanding job of finding and fixing the many bugs in the original prototype code written by the author to explore this topic. The author would especially like to thank Brent Hawks for the long hours he spent serving as a sounding board for, and generator of, ideas. Finally, thanks are gratefully extended to John Gerth for a thorough review of the text and his very helpful suggestions.

Cited references

1. *VS APL Program Logic*, LY20-8032, IBM Corporation (1976); available through IBM branch offices.
2. D. E. Knuth, "Fundamental Algorithms," *The Art of Computer Programming*, Volume 1, Addison-Wesley Publishing Co., Reading, MA (1968).
3. K. Knowlton, "A Fast Storage Allocator," *Communications of the ACM* **8**, No. 10, 623–625 (October 1965).
4. D. S. Hirschberg, "A Class of Dynamic Memory Allocation Algorithms," *Communications of the ACM* **16**, No. 10, 615–618 (October 1973).
5. B. Cranston and R. Thomas, "Simplified Recombination Scheme for Fibonacci Buddy Systems," *Communications of the ACM* **18**, No. 6, 331–332 (June 1975).
6. K. K. Shen and J. L. Peterson, "Weighted Buddy System for Dynamic Storage Allocation," *Communications of the ACM* **17**, No. 10, 558–562 (October 1974).
7. J. L. Peterson and T. A. Norman, "Buddy Systems," *Communications of the ACM* **20**, No. 6, 421–423 (June 1977).
8. G. Bozman, W. Buco, T. P. Daly, and W. H. Tetzlaff, "Analysis of Free-Storage Algorithms—Revisited," *IBM Systems Journal* **23**, No. 1, 44–64 (1984).
9. I. P. Page and J. Hagins, "Improving Performance of Buddy Systems," *IEEE Transactions on Computers* **C-35**, No. 5, 441–447 (May 1986).
10. A. Kaufman, "Tailored List and Recombination-Delaying Buddy Systems," *ACM Transactions on Programming Languages and Systems* **6**, No. 1, 623–625 (January 1984).
11. C. Bays, "Comparison of Next Fit, First Fit, and Best Fit," *Communications of the ACM* **20**, No. 3, 191–192 (March 1977).
12. A. Appel, "Simple Generational Garbage Collection and Fast Allocation," *Software Practice and Experience* **19**, No. 2, 171–183 (February 1989).
13. P. Wilson and T. Moher, "Design of an Opportunistic Garbage Collector," *ACM SIGPLAN Notices* **24**, No. 10, 23–25 (October 1989).

Accepted for publication July 16, 1991.

Ray Trimble *IBM Santa Teresa Laboratory, 555 Bailey Avenue, San Jose, California 95141.* Mr. Trimble is an advisory programmer in APL products development. He has worked in the APL organization since 1976 and has been a technical leader in a number of development projects for both VS APL and APL2. He is the author of the manual *APL2 Programming: Processor Interface Reference* and has been heavily involved in developing other APL manuals. Mr. Trimble joined IBM in 1966 and worked first on a large-scale macro preprocessor supporting multiple languages. Beginning in 1971 he served as joint chief programmer for access methods in the original MVS development project.

Reprint Order No. G321-5446.