CMU-CS-78-111

Evaluation of the Bitstring Algorithm[1]

Peter H. Feiler

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

12 April 1978

Abstract

A resource allocation technique based on an alternative data representation to the list structure, i.e. the bitvector, is discussed in this paper. The data structure provides for implicit collapsing of available resources, and the algorithm, called Bitstring, can be applied to any type of resource without performance loss. An optimized implementation of Bitstring is compared with a corresponding list structure algorithm (Firstfit). Two bitvector algorithms for special resource allocation environments, Exactstring and Quickstring, are presented. The implementation of Bitstring in microcode on a PDP-11/40E and the resulting performance improvement relative to the assembly code implementation are discussed.

---

[1]This work was supported by the Defense Advance Research Projects Agency under contract F44620-73-C-0074, monitored by the Air Force Office of Scientific Research.

1. Introduction

Activities on computer and programming systems change over time. Thus system resources must be managed dynamically. Resources are generally kept in resource pools. Requests for resources are granted by allocation of free resources from the resource pool to the requestor. Upon release of resources by the requestor they are returned to the resource pool. Resource management involves several kinds of resources, eg. primary memory, secondary memory, devices.

Resources may be requested from the pool in units of one at a time, several at a time, or several adjacent units at a time in the case of ordered resources. In the latter case a total ordering of the resource type must exist, as with primary memory addresses or sector addresses of a disk. Each resource name is a unique identifier for a resource within its resource type. A collection of resources is referred to by the set of their resource names or, in the case of ordered resources with adjacent allocation, by the resource name of the first resource and the number of resources in the collection.

Various resource allocation techniques have been developed, choosing different data representations for the resource pool and employing different strategies for allocation and deallocation of resources. In the literature mostly allocation techniques for primary memory are discussed. These dynamic storage allocation (DSA) techniques are based on list structured data representation at no storage cost, because the list elements are kept in the memory space of the free storage. The resource name of free memory blocks is identical to the address of the list element describing the resource. For resources other than primary memory, list elements cannot be embedded in the free resources themselves. List elements must be maintained as separate entities in primary memory. The amount of storage required to represent a dynamically managed resource is then relatively high, and alternative, more compact representations can be thought of.

One such alternative, the *Bitstring* algorithm using a *bitvector* representation [Habermann], is discussed in this paper. Chapter 2 describes the data structure and an optimized algorithm for its manipulation. In chapter 3 its performance is analyzed and compared with corresponding DSA techniques based on list representation. Specialized versions of the Bitstring algorithm for certain allocation environments are given in chapter 4. Finally, implementations of the Bitstring algorithm at different hardware levels, eg. machine code and microcode, are evaluated in chapter 5.

2. Bitstring Algorithm

2.1. The Bitvector

A resource in a resource pool is identified by its *resource name* and an indication of whether it is available (*present*) or allocated (*absent*). The resource pool is represented by a *bitvector*, each bit corresponding to the two-valued state variable of a resource. The length of the bitvector is determined by the total number of resources in the pool. A resource can be referred to in the resource pool by a unique identifier, the *resource index* into the bitvector for the bit representing the state of the resource. There exists a one-to-one mapping from the set of unique *resource names* of resources in a pool onto the set of resource indices of its bitvector.

f: resource name -> resource index.

For example, take the mapping function for the resource type primary memory, where the resource unit is a memory block of n words. The address of the first word of a memory block is converted into the bitvector index by

f (word address) = word address / n = bitvector index .

The bitvector imposes a total ordering on the indices of its resources. Two resources are adjacent if their respective resource indices i,j are adjacent i.e. they differ by one. This ordering is maintained for the resource names if and only if the mapping function f is strictly monotonically increasing or decreasing, and the inverse mapping function f^{-1} is defined everywhere in the range of f and has the same functional properties.

A resource manager may use an instance of the bitvector to represent its resource pool. It defines the mapping function f and its inverse, f^{-1}, in order to perform the appropriate conversions between the resource indices and resource names.

The information about a resource, provided by the bitvector, is often not sufficient for the tasks of a resource manager. Resources can be allocated from the bitvector only to one requestor at a time. Thus sharing of resources cannot be expressed by the bitvector. The operations on the bitvector make assumptions about the proper release of allocated resources, eg. return by the owner and no outstanding references, which must be satisfied by the resource manager, in order to guarantee a consistent representation of the state of the resources in the bitvector. The resource anager maintains additional information about

allocated resources, such as ownership of an allocated resource, number of outstanding references to an allocated resource, and the resource requirements and allocations of different users for deadlock avoidance. An example of a resource manager is the primary memory manager, who keeps segment descriptors with reference counts for allocated memory blocks and a bitvector to represent available memory.

M adjacent resources ($m > 0$) can be allocated in one operation. They are referred to by the resource index of the first resource and the number of resources. When resources are returned to the pool, the bitvector representation causes them to be collapsed with adjacent available resources automatically without additional computation. The list representation does not have this property. In order to avoid continuous splitting of sets of resources, the algorithms on the list represenation execute a collapsing function at resource deallocation time or upon failure to allocate requested resources.

Every resource to be entered or removed from a resource pool is represented by a unique bit in the respective bitvector. If a resource can reside in different pools, it must have a representative bit in each of the corresponding bitvectors. A resource then can be made available in all pools simultaneously, which implies its removal from all pools on an allocation, or it can reside in one pool at a time. The total number of resources managed by a bitvector can be changed dynamically in a restricted form. The bitvector can be extended at either end, i.e. the added resources must be adjacent to existing resources in the pool.

2.2. Basic Algorithm

An allocation request can be made for m adjacent resources: a *resource set* of size m. The request is satisfied if an available resource set of that size can be found. In terms of the bitvector the allocation can be expressed as finding a subvector of size m in the bitvector with all elements *present*. The allocation is recorded by marking all bits of that subvector as *absent* and returning the index of the first bit in the subvector. An allocated set of m adjacent resources is returned to the resource pool by changing the bit values of the corresponding subvector to *present*. For

bitvector = *array* [0 : poolsize-1] *of* bit *initialize* present

we have

```
alloc ( m:requestsize ) returns resourceindex =
        if
                ∃ k ∈ [ 0 : poolsize-1 ] s.t.
                        [ k : k+m-1 ] ⊂ [ 0 : poolsize-1 ]
                ∧ ∀ i ∈ [ k : k+m-1 ] : bitvector[i] = present
        then
                ∀ i ∈ [ k : k+m-1 ] : bitvector[i] ← absent
                result ← k
        else
                result ← exception "request cannot be satisfied"
        fi
```

and

```
free ( p:resourceindex , m:requestsize ) =
        precondition [ p : p+m-1 ] ⊂ [ 0 : poolsize-1 ]
                        ∧ ∀ k ∈ [ p : p+m-1 ] : bitvector[k] = absent

        ∀ k ∈ [ p : p+m-1 ] : bitvector[k] ← present
```

2.3. The Search Algorithm

The bulk of the work in allocation of resources lies in the search for an available resource set of the requested size *m*. We need a strategy to scan the bitvector for a bitpattern of *m* contiguous bits *present*.

A subvector of the requested size - a *window* into the bitvector - is positioned at the left end of the bitvector. It is tested for all *present* bits. If an *absent* bit is encountered, the window is moved to the right of that bit and the test of the window is started over again. The window can be scanned from the left, in which case the leftmost bit *absent* is detected first and the window moved beyond it. If several bits have the value absent, each of them will be encountered and cause the window to be moved. The scan from the left is effectively a linear test of all bits in the bitvector from the left end up to and including the subvector satisfying the allocation condition. The scan of a window from the left is illustrated in fig.1.a. *Lb* and *ub* indicate the lower and upper bound of a window, the subscript *i* specifies the i-th window position in a search, and *cur* the position of the scan head.

If we scan a window from the right, the rightmost bit with the value *absent* is detected and the window moved beyond it. From the illustration in fig 1.b we can observe, that bits may be skipped without being tested, i.e. the bits to the left of the *absent* bit, and that part of the window in the new position has been tested in the previous position of the window, i.e. the

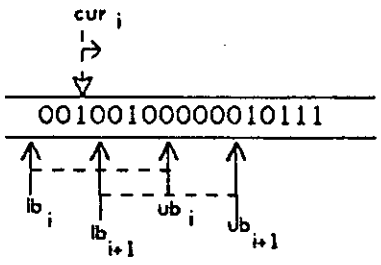

Fig. 1.a

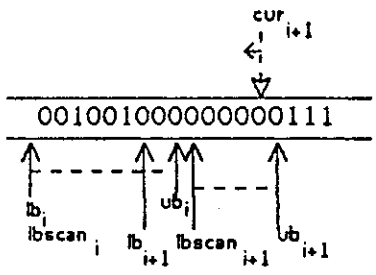

Fig. 1.b

bits from lb_{i+1} to ub_i. Therefore only a subwindow starting at $lbscan_{i+1}$ has to be tested in the (i+1)-th window position. An algorithm for the window test from right is given in A.1.

During the scan in a window test two conditions are checked on every iteration of the scan loop: is the loop variable in the range of the scanned window and is the value of the scanned bit equal *present*. The test of the loop variable can be eliminated as follows. If the bit to the left of the scanned window is guaranteed to have the value *absent*, it is assured that a scan from the right with an infinite loop, testing only the scanned bit, will terminate. The window is successfully scanned, if the rightmost bit detected is the bit to the left of the scanned window. In other words

```
for i from ub downto lbscan do
        if bitvector[i] eql absent then
                return "rightmost absent bit is i"
        fi
od
return "all present bits"
```

can be replaced by

```
with
        savbit = bitvector[lbscan-1]
        i = ub
do
        bitvector[lbscan-1] ← absent
        while bitvector[i] eql present do i ← i-1 od
        bitvector[lbscan-1] ← savbit
        if i eql lbscan-1 then
                return "all present bits"
        else
                return "rightmost absent bit is i"
        fi.
```

If we always start the search with the first window position at the left end of the

bitvector, bitpatterns of values *absent* will tend to cluster at that end. They are skipped at the beginning of each search. An alternative is to maintain a *rotating pointer* which indicates the starting point of the search. We require that the starting point refers to the first element of an available resource set (bitpattern *present*). After a successful search the pointer is updated to refer to the right of the allocated window. Thus a new search will start at the least recently tested position of the bitvector. An algorithm implementing the rotating pointer is given in A.2.

2.4. Hardware Influences

In general machines are not equipped with mechanisms for bit addressing of memory. 8 bit bytes and memory words are the directly addressed memory units. Machine instructions define operations on bytes or words. For the manipulation of individual bits within a word, boolean masking operations, shift operations, and field extraction operations are provided. Some machines have special instructions on individual bits such as branch on the value of the sign bit of a byte or word, or determine the position of the rightmost bit with value 1 .

Keeping the underlying hardware in mind, we may revise the implementation of the bitvector and the allocation and deallocation algorithm. The resource pool can be thought of being implemented as a vector of words, and the resource index of the one-dimensional bitvector is mapped into a two-dimensional bitarray. Tests and assignments of new values can be performed on words (bytes) in one operation. Thus the number of iterations in loops to test or set bit sequences is reduced. However the boundaries of bit sequences require special treatment. Subsets of bits in a word have to be operated on.

The search algorithm must find the exact bit position of the rightmost bit in a window with value *absent*. In the two-dimensional bitarray all bits in a word are tested for presence in one operation. If a word test fails, the position of the rightmost bit absent in that word must be found. This may involve one hardware instruction (such as JFFO on the DEC PDP10 [PDP10]) or more elaborate processing, depending on the underlying machine. The machine of choice is the PDP11-40/E, a 16 bit processor [PDP11] with microcode extension [Fuller]. This machine has byte addressing, and word and byte operations. The wordscan to find the rightmost bit is programmed with shift and mask operations. Three search strategies are worth considering: linear search, binary search, and computed branch. They are discussed in detail in 3.1.6.

3. Evaluation Of Bitstring

3.1. Execution Cost Of The Bitstring Algorithm

3.1.1. Basic Algorithm

The basic algorithm for allocation in a bitvector consists of four parts: preprocessing, search for a window satisfying the allocation condition, change of values in that window, and postprocessing. The search for a valid window is described in more detail as the cost of one window test times the number of window tests in a search. The cost of a window test consists of the overhead for setting up the window and the number of bits to be tested in the window. The cost of marking a window allocated or released is dependent on the size of the window. The cost functions for the two resource allocation operations are

(C1) $$C_{alloc} = C_{allocpre} + C_{search} + C_{windowset} + C_{allocpost},$$

(C2) $$C_{free} = C_{freepre} + C_{windowset} + C_{freepost},$$

(C3) $$C_{windowset} = nbitsinwindow * C_{bitset}$$

(C4) $$C_{search} = nwindowtest * (C_{windowoverhead} + C_{windowtest}),$$

(C5) $$C_{windowtest} = ntestbitsinwindow * C_{bittest}.$$

Nbitsinwindow is the average number of resources allocated in a request. *Ntestbitsinwindow* is the average number of bits actually tested in a given window position. This number is less than *nbitsinwindow*, because the remainder of the window in a window position is not tested when an *absent* bit is detected. *Nwindowtest* is the average number of window positions tested in one search. The product *nwindowtest* * *ntestbitsinwindow* corresponds to the average total number of tested bits in one search.

3.1.2. Scan From Right

We have the choice of scanning a window from the left or from the right. In the scan from the left, the number of times a window is moved is equal to the number of bits absent from the starting point to the position of the valid window. Similarly, the total number of bits tested is the number of bits from the starting point to the first bit of the valid window plus the size of the window. Both statements are obvious by examining the algorithm which is a

linear scan of the bitvector from left through the bitvector, i.e. every bit is tested and every time an absent bit is encountered the window is moved.

A scan of the window from the right may improve the search with respect to both the number of window tests and the number of bits tested in a search. First we show that the number of window tests can be reduced. Given a window position let k be the number of bits absent in the window and p be the position of the rightmost bit absent. A scan from the left will encounter all k bits absent and move the window k times in order to position it to the right of bit position p. In a scan from the right the rightmost bit position p is detected first and the window moved passed the position in one windowtest. For $k = 0$ both the scan from left and from right are successful in one windowtest. For $k > 0$ the scan from right requires $k-1$ windowtests less than scan from left in order to skip to bit position $p+1$.

Next we show that the number of bits tested in a search may also be reduced by a scan from right. We know about scan from left that exactly $w+n-1$ bits are tested once, where w is the first bit position of the valid window and n the windowsize. It will be shown that in a scan from right (algorithm A.1) all bits in [$0:w+n-1$] will be tested at most once and that there may exist bits in that range that are not tested. The first point is clear when recalling that the scanned ranges of two subsequent window tests do not overlap, i.e. $ub_j < lbscan_{j+1}$. In order to show that frequently some bits in the range [$0:w+n-1$] are not tested, it suffices to demonstrate that in any window test some bits are possibly not tested because the scanrange of window tests do not overlap. Given k as the number of bits absent in a window, p as position of the rightmost bit absent, and w as windowsize, we have for p in [$1:w$] that the number of bits tested equals w if $k = 0$, and equals $w-p+1$ if $k > 0$.

Examination of (C4) shows that a reduction of the number of window tests (*nwindowtest*) and of the total number of bittests (*nbitsinwindow*) reduces the cost of allocation.

The scan from right corresponds to the "Fast String Searching Algorithm" [Boyer] with a two element alphabet (*present,absent*) and one search pattern of all present elements of different lengths. An analysis of that algorithm shows that the number of bits inspected is typically less than linear and the worst case behavior is linear in $w+n$. Furthermore it is shown that the number of tested bits decreases with increasing length of the search pattern.

3.1.3. Two-Dimensional Array

The use of a two dimensional bitarray and word operations reduce the number of bit-test and set operations performed in an allocation or deallocation. Given a bitsequence of length n and a wordsize of w bits, n bit operations are replaced by n/w word operations plus additional mask operations at the boundaries of the bitsequence. The cost functions $C_{windowset}/C_{windowtest}$ (C3/C5) can be redefined from

$$C_{1\text{-}dim} = n * C_{bitset/test}$$

to

$$C_{2\text{-}dim} = (n / w) * C_{wordset/test} + 2 * C_{boundary}$$

$C_{boundary}$ is one mask operation. The operation for bitset/test (mask operation) is slower than the wordset/test operation. Thus $C_{1\text{-}dim}$ is greater than $C_{2\text{-}dim}$ for n greater two for any wordsize w. However, the cost is influenced by the available instruction set.

Bitpositions in the bitvector are given by the resource index. Manipulation of the index consists of simple arithmetic. The bitarray requires an indexpair (x_1,x_2) as pointer to a specific bitposition, where x_1 is the wordindex and x_2 is the bitindex within word x_1. Manipulation of this pointer in general requires arithmetic on both indices with overflow from x_2 to x_1. The wordsize of 16 ($= 2^4$) bits on the PDP11 simplifies this operation. The pointer manipulation in a window set/test, which is most frequently executed, only changes the wordindex x_1. The indexpair can be used in two different ways. In the first alternative the resource index is converted to an indexpair at entry/exit of an operation and the indexpair is manipulated in the algorithm. In the second alternative resource indices are manipulated in the algorithm and are converted to indexpairs at access time of the bitarray. A comparison of implementations showed that the first alternative produces more efficient code on the PDP11, because the relatively expensive conversion from resource index to indexpair is executed less frequently.

3.1.4. Finite vs. Infinite Loop

The scan loop in a window test can be implemented as a finite loop. The execution cost $C_{windowtest}$ (C5 for the bitarray) amounts to

$$C_{fin} = n * (C_{wordtest} + C_{loopvarupdate} + C_{loopvarcheck}) + 2 * C_{boundary} +$$

$C_{bitabsent}$,

where n is the number of word tests. C_{absent} is the cost of finding the position of the absent bit in a window test. If the window has all present bits, the cost is zero. If there exists an absent bit, the rightmost absent bit is determined at the cost $C_{findrightmostbit}$.

In the infinite loop implementation the cost of the loop body is reduced by the cost of the loop variable check. However cost incurs from setting the bit to the left of the window, and every window test determines the rightmost bit in a word. Its position is compared with the bitposition to the left of the window in order to determine whether the window had all present bits.

$$C_{inf} = n * (C_{wordtest} + C_{loopvarupdate}) + C_{boundary} + C_{leftbit} + C_{findrightmostbit} + C_{testforsuccess}$$

The finite loop implementation is more efficient if the following relation holds:

$$(n * C_{loopvarcheck} + C_{boundary} + C_{bitabsent}) < (C_{leftbit} + C_{findrightmostbit} + C_{testforsuccess}).$$

This is the case for the BLISS11 implementation of the algorithm as long as the average number of word tests in a window test is less than 5, i.e. less than 80 bits are tested.

3.1.5. Rotating Pointer

The *simple Bitstring* algorithm (A.1) has the characteristics of a Firstfit algorithm. It tends to cluster allocated resources at one end of the resource pool [Knuth]. If the search is always started at that end, the window is moved over all allocated resources. The use of a rotating pointer in the *modified Bitstring* algorithm (A.2) will distribute the allocated resources across the pool, and start the search for free resources at the least recently tested resources. The modified Bitstring algorithm uses the same search strategy as the modified Firstfit algorithm [Weinstock]. The number of window tests per search is drastically reduced (see 3.2.4).

The additional cost for the rotating pointer is minimal, i.e. two words of storage for the two-dimensional pointer, two instructions each for $C_{allocpre}$ and $C_{allocpost}$ for retrieval and update of the pointer, an occasional switch to the left subvector of the search, and a pointer update in *free* if the rotating pointer refers into a bitpattern present after collapsing of adjacent sets of resources.

3.1.6. Rightmost Bit in Word

The PDP11 provides both word and byte operations. Thus the byte containing the rightmost bit absent can be retrieved in one byte test and one byte swap operation. For the search within a byte three strategies are suggested: linear search, binary search, and computed branch.

The *linear search* tests the 8 bits of a byte in linear fashion from right to left. At most seven bits are explicitly tested, because we know that at least one bit in the byte is absent. The position of the absent bit is determined in min. 4 instructions and max. 16 instructions (an average of 10 instructions).

Binary search divides the bitsequence into two parts. The half containing the absent bit is selected for repetition of the process. Three repetitions are necessary to determine the absent bit in a byte. The execution cost is min. 6 instructions and max. 11 instructions (average of 9 instructions). However, relatively expensive ASH instructions are executed which outweigh the gain of one instruction over linear search.

The *computed branch* version for finding the position of the rightmost bit in a byte uses the fact that, given an arbitrary bitsequence X, $S = X \wedge -X$ results in a bitsequence, for which bit p has the value one and all other bits the value zero, where p is the position of the rightmost bit one in X.

Arithmetic on the PDP11 is done in two-complement, ie. $-X = (\neg X) + 1$.

Define $Y = X\,[0{:}p-1]$ and $Z = X\,[p{:}n-1] = 10^{n-p-1}$,ie. $X = Y \circledast Z$.

Thus we get $-X = (\neg Y \circledast \neg Z) + 1 = (\neg Y \circledast 01^{n-p-1}) + 1 = (\neg Y \circledast 10^{n-p-1})$ and

$S = X \wedge -X = (Y \circledast 10^{n-p-1}) \wedge (\neg Y \circledast 10^{n-p-1}) = 0^p 10^{n-p-1}$.

S is used as index into a sparse array to retrieve the bit position. Only the entries with indices 2^n are used where n is the bit position. Computed branch requires 5 instructions. This search strategy takes the least execution time, but requires the implementation of a sparse array with 8 entries. Access cost to the sparse array is not included in the instruction count.

3.2. Comparison with List Structure Representation

3.2.1. Classification

A resource management technique can be characterized by the choice of data representation for the resource pool, and by different allocation and deallocation strategies. [Weinstock] did a study and evaluation of a collection of dynamic storage allocation (DSA) techniques. All DSA algorithms are based on the same representation of the resource pool. Available resources are maintained in linked list structures. For resources such as primary memory linkfields and resource information can be maintained in the storage space of the available resources. This scheme cannot be applied to resources without usable storage such as devices, and storage has to be provided for the list elements by the allocator.

A *model* [Weinstock] describes DSA techniques along several dimensions , such as search criterion, ordering of free resources, collapsing of adjacent resources in an ordering, splitting of resource sets, rounding of requests. The model is powerful enough to describe all dynamic allocation techniques that appeared in the literature. Possible search criteria are:

- first fit - use the first free set of resources larg enough to satisfy the request
- best fit - use the smallest free resource set large enough to satisfy the request
- modified first fit - first fit starting where the previous search left off
- quick fit - combination of several techniques for BLISS11 compiler [Wulf]

among others. Several orderings of free resources are considered.

- LIFO order
- random order
- order by resource name (physical address of primary memory)
- order by size of resource sets (size of free memory blocks)

Collapsing of resource sets may be done at allocation or deallocation for all resources in a given ordering or only for resources previously split.

The simple bitstring algorithm can be described using this model as a resource management technique with first-fit search strategy, ordering of resources by resource name (resource index), and collapsing of adjacent resource sets at deallocation. No rounding is employed and the unused part of a split resource set is returned to the free resource pool. An algorithm

with list structure representation, Firstfit with location ordering and collapsing on free [Weinstock], matches the description of the model for the simple Bitstring because the model does not distinguish between alternative data representations of the resource pool. We call two algorithms *equivalent* in their allocation behavior if they are represented by the same model.

The Bitstring algorithm with rotating pointer uses a different search strategy than the simple Bitstring algorithm. The search in the pool is started where the previous search left off. The equivalent algorithm with list structure representation is the *modified Firstfit* algorithm with location ordering and collapsing at free [Weinstock].

3.2.2. Measures

Resource management techniques can be compared according to one or several measures. Measures are expected to capture different aspects of performance, providing a basis for comparisons. [Shore] uses a *time-memory* product to compare different DSA techniques. This measure does not have an intuitive interpretation for different values on the scale. [Weinstock] tries to circumvent this problem by evaluation of two measures, *resource utilization* and *execution time*. Resource utilization is defined as the probability that a method will be unable to satisfy a request. Resource utilization does not change for equivalent algorithms.

For the comparison of the Bitstring and the Firstfit algorithms we introduce an additional measure in order to reflect different data representations, *memory space* required in the resource manager to implement the resource pool. This gives us a handle on the space efficiency of the encoding of resource information. However the encoding also effects the execution cost. Thus both measures are considered together in an evaluation.

3.2.3. Simulation

The evaluation of the Bitstring algorithm and its comparison with the Firstfit algorithm is based on measurements taken from simulation experiments. The type of the simulated environment affects the performance of a resource allocation algorithm. It is characterized by the number of resources in the pool, the arrival rate of requests for sets of resources, the

size of requested resource sets, and the time span of allocation of the resource set.

Primary memory was chosen as the type of resource in the simulation environment. Few statistics on the rate of memory requests, the size of requests, and the life time of allocated memory are published. [Totschek] obtained statistics from the SDC timesharing system, including distribution of jobsizes. Two reports from the University of Virginia [Batson70,Batson74] publish data on distributions of request size and lifetime of memory blocks from measurements of the ALGOL60 runtime system on a B5000 computer system. The data indicate that most of the requests are for less than 50 memory words. The shape of the lifetime curve is approximately negative exponential. [Weinstock] provides information on distributions of requestsize, lifetime, and interarrival times of requests. The data were collected from the BLISS11 compiler [Wulf]. The distribution of lifetime shows similar behavior for both the Virginia and BLISS11 data. The requestsize for the BLISS11 compiler has an average of less than 10 words. The interarrival time distribution behaves approximately negative exponentially.

For the simulation environment we chose the interarrival time and lifetime distributions from the BLISS11 compiler. The distribution of request size was a uniform distribution with minimum of one and maximum of 50 memory words. The small mean value of the request size suggests a resource pool of 3K words. A simulation run with a larger resource pool showed that the choice of 3K words is sufficient for satisfactory simulation results. The mean value of the lifetime of allocated memory was a variable parameter to the experiment. For evaluation of execution cost the mean lifetime was chosen such that no measured algorithm failed for the period of measurement. Simulation runs started with empty memory. Test runs starting with partially full memory showed no significant influence on data collected from 50,000 events (allocations and deallocations).

Statistics were collected on the number of allocated resources and the average request size. For the list representation the length of the freelist and the number of list operations for allocation and for deallocation were determined. For the bitstring representation the number of window tests and the number of word tests for both the average window test and the setting of bitsequences were measured.

3.2.4. Results

The execution cost of the Bitstring algorithms is calculated using the cost functions (C1) and (C2). Analog cost functions can be defined for the Firstfit algorithm:

$$C_{FFalloc} = C_{allocpre} + avgnlistopsalloc * C_{listop} + C_{allocpost}$$

$$C_{FFfree} = C_{freepre} + avgnlistopsfree * C_{listop} + C_{collapse} + C_{freepost}$$

The measure of space requirement for the bitvector representation depends linearly on the total number of resources in the resource pool. For the simple Bitstring algorithm we have

$$Space_{bitstring} = nresources \text{ [bits] } = nresources \ / \ wordsize \text{ [words]}$$

The freelist of the list representation requires two linkfields for the listhead and three fields (two links and a size) for each list element representing an available set of resources. As cost function we have

$$Space_{firstfit} = Space_{head} + nelements * Space_{listelement}$$

where the cost of a list element is zero if it can be embedded in the storage of a free resource set. The number of list elements depends on the application environment. The maximum number of nonadjacent sets of free resource sets possible in a resource pool is an upper bound on the size of the freelist. The smallest resource set is of size one, and free resource sets have to be separated by allocated resource sets. Thus the maximum freelist length is *total number of resources / 2*. In general the freelist is considerable smaller. According to Knuth's fiftypercent rule [Knuth] for Firstfit with collapsing, the number of free resource sets is $1/2 * p * n$, where n is the number of allocated resource sets and p is the probability that all sets of free resources are bigger than the requested size. P is 1 when the block sizes are infrequently equal to each other. In that case the number of free resource sets is half the number of allocated resource sets.

Simulation Statistics

number of events	30000	40000	50000
number of alloc operations	15019	20020	25019
number of free operations	14981	19980	24982
avg. number of allocated resource sets	38	40	39
avg. number of allocated resources	1145	1157	1156

First-Fit

avg. number of list operations in alloc	9.4	9.4	9.4
avg. number of list operations in free	10.8	10.8	10.8
avg. freelist length	19.8	19.8	19.8

Simple Bitstring

avg. number of windowtests	21.0	21.1	21.1
avg. number of wordtests in window	0.11	0.11	0.11
avg. number of wordsets in window	0.77	0.77	0.77

Modified Bitstring

avg. number of windowtests	4.3	4.3	4.3
avg. number of wordtests in window	0.46	0.46	0.46
avg. number of wordsets in window	0.80	0.80	0.80

Execution Cost (in number of instructions)

First-Fit

C_{alloc} = 15 + 5 * nlistopsinalloc = 62
C_{free} = 36 + 5 * nlistopsinfree = 90
C_{avg} = (C_{alloc} + C_{free}) / 2 = 76

Simple Bitstring

C_{alloc} = (3 * nwordtestinwindow + 36) * nwindowtest +
3 * nwordsetinwindow + 65 = 833
C_{free} = 3 * nwordsetinwindow + 30 = 32
C_{avg} = 434

Modified Bitstring

C_{alloc} = (3 * nwordtestinwindow + 37) * nwindowtest +
3 * nwordsetinwindow + 69 = 234
C_{free} = 3 * nwordsetinwindow + 32 = 34
C_{avg} = 134

Comparison of Execution Cost

Average

First-Fit/Mod. Bitstring = 76 / 134 = 1 : 1.76 = 0.56
First-Fit/Simple Bitstring = 76 / 434 = 1 : 5.71 = 0.17
Mod.Bitstring/Simple Bitstring = 134 / 434 =1 : 3.23 = 0.30

Alloc

First-Fit/Mod. Bitstring = 62 / 234 = 1 : 3.77 = 0.26

Free

First-Fit/Mod. Bitstring = 90 / 34 = 1 : 0.35 = 2.8

The use of the rotating pointer in the Bitstring algorithm reduces the average number of window tests considerably (from 25 to 4). Knuth's fifty-precent rule is confirmed with an average of 40 allocated memory blocks and 20 free memory blocks. The execution time clearly indicates that the bitvector representation is more expensive. The best Bitstring algorithm, the modified Bitstring, is on the average 1.76 as slow as the Firstfit algorithm. Note, however, that the deallocation function by itself is less expensive for Bitstring. The collapsing of adjacent free resource sets does not require any computation. This is an inherent property of the chosen representation.

For the modified Bitstring algorithm the space requirement for 3K resources is $2 + 3 * 2^6 = 194$ words. The Firstfit representation requires 2 words, if the list elements are stored in the free resources, and an average of 62 words for the simulated environment. However, the worst case behavior of the list representation requires a maximum of 4.5K words of storage.

4. Quickstring Algorithm

In some resource management environments, like the dynamic memory management of the BLISS11 compiler, standard allocation techniques had shown to be unreasonably slow [Wulf]. In 1971 Wulf, Weinstock, and Johnsson designed and implemented the Quickfit method for the BLISS11 compiler, taking advantage of the observation that in certain environments allocation requests dominate for a small set of sizes. This method has been evaluated and compared with other DSA techniques with list representations [Weinstock] and shown to perform superior to them in most cases. A similar algorithm can be developed for the bitvector representation.

4.1. Exactstring

Allocation requests of one size only are a special case of an allocation environment for the Quickfit method. Resource sets of the requested size are represented by one state bit. Thus allocation of resources consists of finding one present bit. The resource pool *bitpool* is implemented as a two-dimensional array of bits. The search for an available resource consists of finding a word in the array with at least one bit with value present. The position of one of these bits (the rightmost one) is then to be determined and its value changed. A specification for Exactstring is given in A.3.

The execution cost for this algorithm is considerably less than for the simple Bitstring algorithm. The cost depends on the number of wordtests performed i.e.

$$C_{alloc} = C_{pre} + avgnwordtest * C_{wordtest} + C_{findbit} + C_{post}.$$

The use of a rotating pointer referring to the last tested word will keep the average number of wordtests well below two, such that the execution time for alloc is about 28 instructions and 7 instructions for free (average of 18 instructions).

The list representation of the resource pool requires one list element per available resource set. The available resource sets are kept in arbitrary order and no collapsing of adjacent resource sets is performed. An implementation of the Exactfit algorithm on the list representation of Firstfit (doubly linked list) results in an execution cost of 14 instructions for alloc and 9 instructions for free (average of 12 instructions). A singly linked list, however, is sufficient to implement a resource pool for Exactfit. The pool operations can then

be performed in three instructions for both alloc and free.

4.2. Quickstring

The collection of resources is divided into pools containing resource sets of the preferred request sizes and one pool for arbitrary sized allocation. Each of the preferred size pools is implemented as a bitpool and the general pool is represented as a bitvector. A resource resides in exactly one pool and cannot migrate to different pools.

The search algorithm of the Quickstring method for a request of size k is as follows:

If the requested size k matches one of the sizes represented by the bitpools

- try to allocate from the bitpool with resource sets of size k
- if failing, try to allocate from the bitvector (general pool)
- if failing, and if there exists a bitpool with larger resource set size m then try to allocate from bitpool of the size m resource sets, wasting $m-k$ resources.

If the requested size k does not match one of the bitpools

- try to allocate from the bitvector
- if failing, and there exist bitpools with larger resource set size m then try to allocate from bitpool of size m resource sets, wasting $m-k$ resources.

The program for Quickstring is given in A.4.

If the resources are divided into the different pools according to the distribution of the request sizes, a high precentage (about 80% for the BLISS11 compiler [Weinstock]) of the requests can be allocated by application of the Exactstring algorithm. Most of the remaining requests are allocated from the general pool via the Bitstring algorithm. Only a small percentage (3% for the BLISS11 compiler) of the requests are retried on several pools. Thus the Quickstring algorithm runs to 80% at the cost of Exactstring (average 18 instructions) plus the overhead of pool selection, and 17% are executed at the speed of modified Bitstring (average 134 instructions) plus the overhead of pool selection.

The Quickfit algorithm described in [Weinstock] differs from the Quickstring algorithm above by returning unused resources of an allocation from a preferred pool to the general pool. The memory utilization of the Quickfit algorithm is very close to that of the Firstfit or

Bestfit algorithm [Weinstock]. Preferred requests of small size cause little additional internal fragmentation to the Quickstring algorithm. In order to make the Quickstring algorithm equivalent to the Quickfit algorithm, we extend the general pool to contain all managed resources. This allows resources to migrate between preferred pools and the general pool. An increase in execution cost is due to migration of unused resources to the general pool in the allocation operation, and a check in the deallocation operation, whether a release of resources is to a preferred pool with unused resources in the general pool.

5. Bitstring in Microcode

An algorithm can be implemented on different hardware or hardware levels more or less efficiently. The DEC-PDP11 has a large instruction set with a powerful addressing mechanism, but lacks sophisticated bit manipulation at the assembly code level. The standard model DEC-PDP11/40 is a horizontally-encoded microprogrammable processor with microcode in read-only memory for the standard instruction set. In a project at Carnegie-Mellon University [Fuller], the processor hardware has been extended by additional functional units to overcome certain deficiencies, and by a writeable micro control store to provide for microprogramming by general programmers. For a description of this processor, the PDP11/40E, the reader is referred to [Fuller]. A simplified data path diagram is given in A.5. Some functional units are explained, as they are relevant to the discussion of microcoding the Bitstring algorithm.

5.1. Improvements of the Implementation

The assembly code and the microcode instruction set each defines a *virtual* processor with different characteristics. The implementations of an algorithm on the two processors are quite distinct.

The starting point for the implementation of the Bitstring algorithm is a high level description of the algorithm in hardware independent terms. Some optimizations, that are inherent to the algorithm and the chosen data structures, are performed at this stage without any knowledge of the underlying processor, eg. scan from right, rotating pointer.

The bit is the smallest data structure supported by the virtual processors, but memory and processor registers are defined in terms of machine words, and operations are executed on words as well as on bits. This fact is used to improve the implementation of Bitstring for both processors, eg. bitvector represenation as two-dimensional array.

The evaluation of alternative algorithms, eg. search for rightmost bit, with respect to execution time is strongly dependent on the underlying processor. The hardware is controlled by means of the instruction set. The assembly processor provides for sequential flow of control, instruction by instruction. All internal functional units complete their operation at the end of every instruction. Thus the processor represents one functional unit to the

programmer. The micro processor has additional internal functional units and makes all functional units directly accessible. They can be utilized simulaneously and partly asynchronously. For example, the implementation of the linear search is superior to binary search on the assembly processor, whereas for the micro processor the inverse is true.

5.2. Micro Processor Usage

The program on the micro processor is located in a fast small memory, the micro control store. The access time is considerably less than access to primary memory. Furthermore, the execution of one instruction overlaps with the fetch of the next instruction, which requires, that a conditional branch has a delay of one instruction, i.e. the condition influencing the branch is temporarily kept in the microcode interpreter before its effect is visible to the program. This implicit buffer is used in the Bitstring microcode to implement access to an array of constants (bitmasks) from several places in the program. Both the array index and the return address are buffered in sequence as branch conditions. They are available to the array access code in the same order.

```
        arrayindex := TOS $;            ! define top of stack to be var.

        case arrayindex<3:0>            ! index to 16 elem. array
        eubc ← retadr; goto array       ! store return address
retadr: <...>                           ! continuation of program

array:      set
            b ← 000001; goto 0          ! get value and return
            b ← 000003; goto 0
            ...
        tes
```

This example also shows the use of the hardware extensions: the stack as storage and input to the mask and shift unit, the mask and shift unit for field extraction, and the n-way branch (EUBC). The n-way branch accepts only constant values from the data field of a micro instruction (eg. *retadr*) or values from the top of the stack, modified by the mask and shift unit. The branch on the D-register being equal to zero is the only other generally useful conditional branch mechanism.

Variables, that require field extraction via the mask and shift unit, can be stored permanently on the stack, only if they adhere to strict scope rules, i.e. only the innermost variable can be manipulated. The implementation of Bitstring is able to keep two variables on

the stack. All other variables are stored in the general registers, and use the top of stack temporarily as working location for field extraction. The stack is also used as a register save area at the entry of a microcode routine, where up to four general registers are preserved for the duration of a Bitstring operation.

The bitvector can potentially be maintained in the micro store extension rather than in primary memory. Unfortunately, the access mechanism is awkward to use. The number of callsites is limited, but may be extended using a subroutine calling sequence, with the result of slower access than primary memory access. Micro store access without subroutine call improves the execution cost of the Bitstring algorithm by a negligible amount (3%).

Even though multiple functional units are under direct control of the microprogrammer, their simultaneous use is limited by bottlenecks in the data flow path and hardware constraints. Transactions on the UNIBUS require, that the input values are stable. Thus, no write operation and about half of the read operations do permit parallel execution of other instructions. Similarly, the general registers are clocked to both the RD-bus and the DMUX-bus in one instruction. Therefore, two different general registers cannot be operated on in one instruction (eg. *ub←d;d←lb+b*, where *lb* and *ub* are general registers). Also, the use of both buses with access to a general register from one bus only produces side effects, such as change of register contents (eg. *tos←d;ba←r1*) or unintentional ORing of a register onto the RD-bus (eg. *r1←unibus;d←tos+2*).

5.3. Comparison of Assembly Code and Microcode

The Firstfit algorithm executes on a small number of variables with few independent operations, i.e. it has little potential parallelism. Its microcode implementation can take little advantage of overlapping operations in the instruction sequence. In addition, the field extraction mechanism is not used. Most of the gain in performance, compared to the assembly code implementation, results from the faster instruction fetch from the micro store and from the maintenance of intermediate execution state between instructions in internal registers (D or B register, stack). The result is a speedup factor in execution time of about 5.

The Bitstring algorithm maintains more computational state information with a high potential of independent operations. A careful analysis of the set of independent operations and of the restrictions on the simultaneous use of different hardware components permits a high degree

of operation overlap, because all functional units are used frequently. The simple Bitstring algorithm with infinite loop and linear search for the rightmost bit was implemented on both the assembly and the micro processor. A comparison showed a speedup of 9.5 for the microcode, about twice the speedup achieved for Firstfit. Thus the microcode version of Bitstring is faster than the microcode version of Firstfit.

The microcode implementation of Bitstring was modified in order to make more effective use of the field extraction mechanism in the search for the rightmost bit. The linear search was replaced by a binary search from 16 bits to 4 bits, and indexing of the 4 bit value index into an array (conditional branch). This implementation is compared with the assembly code implementation with the following results, achieving a speedup of 10.3 - 10.8:

Execution Cost

Assembly (in instructions)

C_{alloc} = (2 * avgnwordtestinwindow + 38) * avgnwindowtest +
3 * nwrodsetinwindow + 80
C_{free} = 3 * nwordsetinwindow + 30
Average instruction time = 2.2 microsec

Micro Code (in microseconds)

C_{alloc} = (1.4 * avgnwordtestinwindow + 7.8) * avgnwindowtest +
1.3 * nwordsetinwindow + 8.8
C_{free} = 1.3 * nwordsetinwindow + 5.5

Execution Time (in microseconds)

	Average	Alloc	Free
Assembly			
Mod. Bitstring	286.0	497.2	74.8
Simple Bitstring	945.7	1821	70.4
Microcode			
Mod. Bitstring	26.3	46.1	6.54
Simple Bitstring	92.0	177.6	6.50

EVALUATION OF THE BITSTRING ALGORITHM

Comparison of Execution Time

Microcode/Assembly		
Mod. Bitstring	1 : 10.8	1 : 10.7
Simple Bitstring	1 : 10.3	1 : 10.2

6. Conclusions

A resource allocation technique based on the use of a bitvector as data representation, Bitstring, has been examined. An optimized algorithm, the modified Bitstring algorithm, was developed for general allocation of an ordered set of resources. The algorithm is compared to resource allocation algorithms which are equivalent with respect the management of resources, eg. search strategy, but are based on a different, list structured representation of the resource pool. The Bitstring is shown to be inferior to the Firstfit with respect to dynamic storage allocation in both execution time (by a factor of 1.75) and space requirement. For resources in general, eg. devices, secondary storage, the Firstfit algorithm has a higher worst case space requirement. For certain resource allocation environments improved algorithms on the bitvector representation are given. An Exactstring algorithm allocates resources of one request size only. For allocation environments with a dominating range of request sizes the Quickstring algorithm is provided, handling those request sizes as special cases. Its search strategy is similar to the Quickfit strategy for list representations [Weinstock]. Implementation of the Bitstring algorithm in both machine code and microcode are considered. The PDP11/40E, a microprogrammable modified DEC-PDP11/40, is taken as the hardware basis. The microcode permits better exploitation of the available hardware at the cost of higher programming complexity. The implementation of the algorithm in microcode results in a speedup factor of about 10, executing faster than the microcode version of Firstfit. The use of microstore as data area is considered, but the performance improvement is negligible, due to hardware constraints.

7. References

[Batson70] Batson,A.P., S.M. Ju, and D.C. Wood, "Measurements of Segment Size", CACM, 13(3), March 1970, 155-159

[Batson74] Batson,A.P., and R.E. Brundage, "Measurements of the Virtual Memory Demands of Algol-60 Programs", Department of Applied Mathematics and Computer Science, University of Virginia, 1974

[Boyer] Boyer,R.S., and J.S. Moore, "A Fast String Searching Algorithm", CACM, 20(10), Oct. 1977, 762-772

[Fuller] Fuller, S.H., etal., "PDP-11/40E Microprogramming Reference Manual", Carnegie-Mellon University, Technical Report, Jan. 1976

[Habermann] Habermann, A.N., "Introduction to Operating System Design", Science Research Associates, Inc., 1976

[Knuth] Knuth, D.E., "Fundamental Algorithms", Addison Wesley, 1973

[PDP10] DEC PDP-10 Assembly Handbook, Digital Equipment Corporation, 1972

[PDP11] DEC PDP-11/40 Processor Handbook, Digital Equipment Corporation, 1973

[Shore] Shore J.E., "On the External Storage Fragmentation Produced by First-fit and Best-fit Allocation Strategies", CACM, 18(8), Aug. 1975, 433-440

[Totschek] Totschek, R.A., "An Empirical Investigation into the Behavior of the SDC Timesharing System", Report SP2191, Systems Development Corporation, 1965

[Weinstock] Weinstock, C.B., "Dynamic Storage Allocation Techniques", Carnegie-Mellon University, Ph.D. Thesis, April 1976

[Wulf] Wulf, W.A., R.K. Johnsson, C.B. Weinstock, S.O. Hobbs, and C.M. Geschke, "The Design of an Optimizing Compiler", American Elsevier, 1975

8. Programs

The appendix of this document only contains the algorithms for simple Bitstring, modified Bitstring, Exactstring, and Quickstring in Type description. The actual implementation of the diverse algorithms in BLISS11 and MICRO40 are not included for space reasons. Copies of the BLISS11 code for the simple and modified Bitstring, Exactstring, Firstfit, and Exactfit algorithms plus the simulation program, and the microcode implementation of the simple Bitstring and the Firstfit algorithms are available upon request.

8.1. Simple Bitstring

```
type bit = ( present , absent )

type bitindex = integer in range of [ bitvectorsize ]


type bitvector [ bitvectorsize ] =
        array [bitvectorsize] of bit


op free ( bitvec:bitvector , bitndx:bitindex , size:integer ) =
        precon bitndx+size-1 in bitindex
              and
              forall i in [ bitndx..bitndx+size-1 ] sat bitvec[i] eql absent

        forall i in [ bitndx..bitndx+size-1 ] do
              bitvec[i] ← present



op alloc ( bitvec:bitvector , nunits:integer ) returns bitindex =

        precon nunits:bitindex

        with
        lb = lowerbound of bitvec          /* lower bound of window */
        ub = lb + nunits-1                 /* upper bound of window */
        lbscan = lb                        /* lower bound subwindow */
        do

        while
              if ub not in bitindex then ERROR "no memory"
              forsome i in -[lbscan..ub] sat bitvec[i] eql absent
        do
              lb ← i+1
              lbscan ← ub + 1
              ub ← lb + nunits-1
        od

        forall i in [ lb..ub] do
              bitvec[i] ← absent
        return lb
```

8.2. Modified Bitstring

```
type rotbitvector [ bitvectorsize ] =
        bv = bitvector [bitvectorsize]
        rotptr = bitindex init lowerbound of bv



op free ( bitvec:rotbitvector , bitndx:bitindex , size:integer ) =
        precon bitndx+size-1 in bitindex
               and
               forall i in [ bitndx..bitndx+size-1 ] sat bitvec.bv[i] eql absent

        forall i in [ bitndx..bitndx+size-1 ] do
               bitvec.bv[i] ← present

        if bitndx+nunits eql bitvec.rotptr then
               bitvec.rotptr ← lowerbound of bitvec.bv
        fi



op alloc ( bitvec:rotbitvector , nunits:integer ) returns bitindex =

        precon nunits:bitindex

        with
        lb = bitvec.rotptr                  /* lower bound of window */
        ub = lb + nunits-1                  /* upper bound of window */
        lbscan = lb                         /* lower bound scanned window */
        maxsize = upperbound of bv          /* upper bound of bv */
        do

        while
               while ub gtr maxsize
               do
                      if maxsize eql bitvec.rotptr then
                             return ERROR "no memory"
                      else
                             maxsize ← bitvec.rotptr-1
                             lb ← lowerbound of bitvec.bv
                             ub ← lb + nunits-1
                             lbscan ← lb
                      fi
               od
```

```
        forsome i in -[lbscan..ub] sat bitvec[i] eql absent
do

        lb ← i+1
        lbscan ← ub + 1
        ub ← lb + nunits-1
od

forall i in [ lb..ub] do
        bitvec[i] ← absent
return lb
```

8.3. Exactstring

```
type bit = ( present , absent )

type word = array [ 16 ] of bit init ( 16: present )


op allabsent ( wd:word ) = returns boolean
        return all i in range of wd sat wd[i] eql present


op findpresent ( wd:word ) = returns wordindex
        precon not allabsent(wd)

        for some i in range of wd sat wd[i] eql present do
                return i



type bitpool [ n ] =
        array [ n / 16 + 1 ] of word


op alloc ( bp:bitpool ) = returns bitindex

        if forsome i in range of bp sat
                not allabsent(bp[i])
        then
                j ← findpresent ( bp[i] )
                bp[i,j] ← absent
                return i*16+j
        else
                return ERROR "no memory"



op free ( bp:bitpool , bpndx:bitindex ) =

        precon bp[ bpndx/16 , bpndx mod 16 ] eql absent

        bp[ bpndx/16 , bpndx mod 16 ] ← present
```

8.4. Quickstring

```
type quickstring [ (reqsize_1,...,reqsize_n),(nresourceset_1,...,nresourceset_n),
        ngeneralresources ] =

    for i in [ 1:n ] : bp_i = bitpool [ nresourceset_i ]
    syn generalpool = bp_n+1 = bitstring [ ngeneralresources ]

    constant base_1 = 0
    for i in [ 1:n ]: base_i+1 = base_i + reqsize_i * nresourceset_i


op alloc ( qs:quickstring , size:integer ) = returns resourceindex

    with bitndx = bitindex do

        if forsome i in [ 1:n ] sat reqsize_i eql size then
            if ( bitndx←alloc(bp_i) ) neq ERROR then
                return base_i + reqsize_i * bitndx

        if ( bitndx←alloc(generalpool,size) ) neq ERROR then
            return base_n+1 + bitndx

        forall i in [ 1:n ] sat reqsize_i gtr size do
            if ( bitndx←alloc(bp_i) ) neq ERROR then
                return base_i + reqsize_i * bitndx

        return ERROR "no memory"


op free ( qs:quickstring , resndx:resourceindex , size:integer ) =

    if forsome i in [ 1:n ] sat resndx ∈ [ base_i:base_i+1-1 ] then
        free ( bp_i , (resndx-base_i)/reqsize_i )
    else
        free ( generalpool , resndx-base_n+1 , size )
    fi
```

8.5. PDP-11/40E Data Path Diagram