International Journal of Parallel Programming, Vol. 17, No. 4, 1988

Algorithms for Parallel Memory Allocation[1,2]

Carla Schlatter Ellis[3] and Thomas J. Olson[4]

Received August 1988; revised February 1989

Dynamic storage allocation is a vital component of programming systems intended for multiprocessor architectures that support globally shared memory. Highly parallel algorithms for access to system data structures lie at the core of effective memory allocation strategies as well as solutions to other parallel systems problems. In this paper, we investigate four algorithms, all based on the first fit approach, that provide different granularities of parallel access to the allocator's data structures. These solutions employ a variety of design techniques including specialized locking protocols, the use of atomic fetch-and-Φ operations, and structural modifications. We describe experiments designed to compare the performance of these schemes. The results show that simple algorithms are appropriate when the expected number of concurrent requests per memory is low and the request pattern is not bursty. Algorithms that support finer granularity access while avoiding locking protocols are successful in a range of larger processor/memory ratios.

KEY WORDS: Concurrent data structures; first fit memory allocation; parallel algorithms; shared memory multiprocessing.

1. INTRODUCTION

Dynamic storage allocation is a vital component of programming systems intended for multiprocessor architectures that support globally shared

---

[1] This research was supported in part by the National Science Foundation under Grant Number DCR 8320136, DARPA/U.S. Army Engineer Topographic Laboratories under contract number DACA76-85-C-0001, and Unisys Corporation.

[2] A preliminary version appeared in International Conference on Parallel Processing, August 1987.

[3] Department of Computer Science, Duke University, Durham, North Carolina 27706.

[4] Computer Science Department, The University of Rochester, Rochester, New York 14627.

0885-7458/88/0800-0303$06.00/0 © 1988 Plenum Publishing Corporation

828/17/4-1

memory. As the number of processors in such machines grows significantly, the development of highly parallel allocation mechanisms becomes increasingly important. In this paper, we investigate four algorithms, all based on the first fit approach,[1] that provide different granularities of parallel access to the allocator's data structures. We describe experiments designed to compare the performance of these four schemes.

In essence, this is a concurrent data structure problem focusing on the parallel manipulation of the list that represents the available blocks of storage. There is a substantial body of literature, known primarily within the database community, that deals with specialized concurrency control algorithms for various search structures.[2–7] The need for better performance in providing operating system services for tightly-coupled multiprocessors suggests that similar techniques must be adapted for parallel access to system data structures (e.g. freelists, priority queues, and mapping tables).

To our knowledge, the problem of parallel memory allocation has not been given the attention it deserves. Stone[8] describes an algorithm for first fit allocation using the fetch-and-add instruction. This solution uses what are essentially read and write locks placed on the freelist at the granularity of individual blocks. It resembles two of our solutions in its use of the fetch-and-add instruction and locking granularity. However, it is based on a different list structure (unordered) and different lock semantics. The paper fails to specify the details of how to search and modify the queue of free blocks (not a trivial aspect of the problem). It also gives no measure of the increase in external fragmentation over sequential first fit or of the increased length of search paths that intuitively should result from conflicts between processes in this approach. These are some of the questions that our project addresses. A parallel version of the buddy system using fetch-and-add has also been developed.[9] A study comparing the performance of three distinctly different allocation schemes on a small-scale multiprocessor is described in Ref. 10. Two of the algorithms considered allow absolutely no parallelism and so, the results are not very informative. In addition, we are interested in multiprocessing environments that scale to many more processors. Finally, Ford[11] has proposed algorithms for dynamic memory management inspired by optimistic concurrency control techniques.[12] Unfortunately, the performance analysis in Ref. 11 does not capture what the dynamic behavior of the optimistic allocation scheme might be on a medium to large-scale parallel processor.

The model of computation assumed in our algorithm design is a shared memory MIMD multiprocessor with atomic fetch-and-Φ instructions.[13] This model is fairly general; it can encompass architectures with different memory/processor configurations (separate memory modules or

memories residing at processor nodes) and different communication technologies (busses or switching networks, with or without combining). Concrete examples of machines that satisfy these requirements include the BBN Butterfly[(14)] and the IBM RP3.[(15)]

The target environment for experimental evaluation of our algorithms is the Butterfly family of multiprocessors built by Bolt, Beranek, and Newman. Experiments have been run on 64 nodes of the 120-node Butterfly[(14)] at the University of Rochester and on the 64-node Butterfly Plus[(16)] at Duke University. These machines are classified as shared memory MIMD nonuniform memory access (NUMA) architectures. The two designs are basically similar: A system consists of up to 256 processor nodes, based upon MC680x0 microprocessors, connected by a switching network that provides logarithmic time communication between any two processors. All memory in the machine resides on the individual nodes, but any processor can access any memory location through the switch. The major difference between the two designs lies in the memory architecture; in particular, the Butterfly Plus has a much wider gap between local and remote memory access times than the original Butterfly.

The motivation for this work grew out of experience with the Uniform System package,[(17)] the most heavily used programming system for the Butterfly. The Uniform System provides tools for process management and allows processes to share a large common address space. It provides an application program with mechanisms to spread its sharable data throughout the machine. Within the Uniform System approach, there is usually only one process per processor. Shared storage is obtained through calls to the memory allocator which, at the time this project began, did not allow any concurrent access. Thus, one of the goals of this work has been to demonstrate the potential benefits of an allocation scheme that offers more parallelism and influence the development of future systems. In fact, recent releases of the Uniform System have adopted an improved algorithm very similar to the one that our experiments show should be the best in this environment.

In the next section, we present each of the four different algorithms under consideration. This includes details of the freelist structure used in those solutions and pseudocode for the more complex algorithms. In Section 3, we describe the design of the experiments used to evaluate the performance of these algorithms and the results of those experiments. Finally, we conclude with observations about algorithm design techniques for this particular class of problems.

2. ALGORITHMS

In this section, we present a number of different approaches to parallel first fit memory allocation. The interface to the memory allocator consists of two calls, **Allocate** and **Deallocate**. In response to an allocation request, the system gives the caller a pointer to a contiguous region of memory of the exact size requested. Deallocation returns a region of memory to the freelist used by the allocator. The goal of our algorithms is to support concurrent allocate and deallocate requests generated by the multiple processors sharing the memory pool.

In order to guide and evaluate alternative design decisions, it is helpful to understand the usage patterns of these calls. Assumptions made about the relative frequency of use determine which aspects of the operations are important to optimize. The distribution of the sizes of requested blocks also influences design decisions, especially with regard to the storage of data used by the allocation routines to describe a free block and maintain the freelist structure. Typically, such bookkeeping information requires several words of storage. If allocation requests tend to be large, then this information can be stored within the free blocks rather than in an auxiliary data structure.

We have not yet done a detailed study to characterize actual usage patterns. However, we have informally surveyed applications programs that have been developed at Duke and the University of Rochester (e.g. circuit simulation, image analysis, and numerical analysis code) and found that applications programs tend to have either a few very large allocations (e.g. an entire shared matrix) or, more commonly and successfully, several moderate size allocations (e.g. the rows of a scattered matrix, <2K bytes). By contrast, programs designed to experiment with parallel algorithms contain many allocations under 64 bytes (e.g. instrumentation variables, linked list records, and locks), but these examples are not considered typical. The application code serves as our model for the distribution of request sizes, with emphasis on the moderate range. Until the nature of the true workload of a particular system is discovered, any recommendations about which algorithms are to be preferred may turn out to be inappropriate for that environment.

For this study, we focus on usage patterns within a single multiprocess application. One commonly observed pattern is for an application to make numerous allocation requests during its execution and then release memory all at once at termination. This implies that deallocations (if done individually) will come in clusters and may significantly compete among themselves for access to the freelist; whereas allocations may be spread over a longer period of time. This clustered pattern of allocations and dealloca-

tions has been observed in most Uniform System applications. This may at least partially be an artifact of early implementations of the Uniform System in which **freeAll** was the only construct available for deallocating storage. The prevalence of this style implies that it may be worthwhile to consider ways of reducing the degree of interaction among concurrent deallocate requests (e.g. deferring the coalescing of adjacent free blocks). Allocations in practice also often occur in one burst at initialization time. For these patterns, allocation and deallocation phases rarely interfere with each other.

A likely alternative pattern is to have an equal number of allocate and deallocate requests scattered throughout the execution of the program (with the obvious constraint that space must be allocated before it can be released). In this case, there is no basis upon which to optimize a particular interaction between operations, but concurrency between allocation and deallocation operations is more important than in the previous case.

2.1 Freelist Data Structure

Available memory is represented as a set of linked lists. Each list represents a pool of contiguous memory locations within a single memory module from which allocations can be made. The nodes of these list structures describe blocks of contiguous free memory and they are ordered by address of the starting location of the block. Initially, each list consists of a single large block. Simple calculations and comparisons on addresses can be used to find neighboring blocks and decide whether two free blocks are adjacent. Associated with each block is header information including *size*, the *next pointer*, and whatever additional bookkeeping is required by a particular allocation scheme. The next field of the last block is *nil.* For most of this discussion, the header data are assumed to reside within the free block itself. Under this assumption, the nodes of the freelist are synonymous with the blocks of available memory. This implies a minimum length allocation unit that is larger than one word. The freelist has an array of designated nodes called *free.* Each element of free appears to be a block with size of zero and physical location not adjacent to any location in the memory pool it controls. These characteristics assure that it can never be allocated or merged with another block. Figure 1 gives an example of a freelist with three lists, ten free blocks (white) and eight allocated regions (shaded).

The purely sequential procedures for the allocate and deallocate operations serve as a basis for the parallel solutions being developed. These sequential operations are briefly outlined as follows.

The procedure for allocating a region of size n is to choose a list and

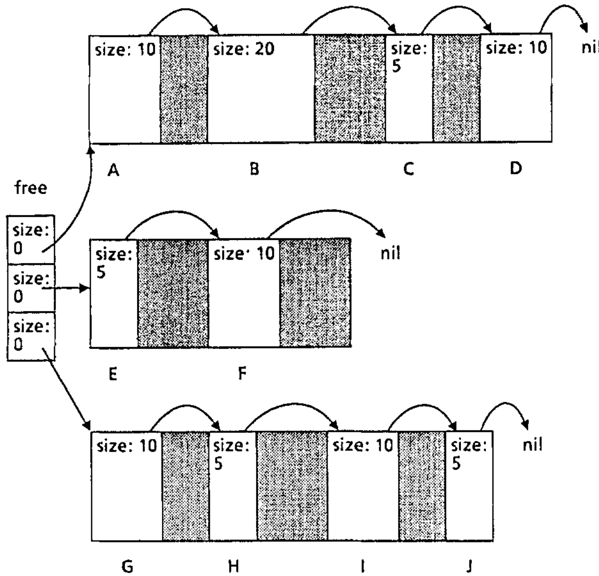

Fig. 1. Freelist data structure.

search it until a block of sufficient size or the end of the list is encountered. In this study we assume that the choice is random although an actual system may support other policies for choosing a list (e.g. allowing the user to explicitly control the site at which an allocation attempt should be made or trying to scatter allocations as evenly as possible throughout the machine).

In our approach, if the request is not satisfied within the randomly chosen list, a retry is performed using another list. At each node of the structure, a routine called **TryToAllocate** is executed which returns a pointer to an appropriately sized chunk if one can be cut off the end of the current block. If the entire free block is needed (i.e. size of current block is n), then its node is removed from the freelist by **DeleteAfter**. Figure 2 shows two allocation requests directed to list i. The initial state for the first call (i.e. a request for 5 units) is given as Fig. 2a which is the same as free[0] of Fig. 1. Figure 2b shows a chunk of size 5 removed from block A leaving a block of size 5. Next, suppose a request for 20 units is made. Figure 2c shows the **TryToAllocate** routine leaving a block of size zero when the entire block B is required and then a call to **DeleteAfter** removes the node from the list.

To deallocate memory, the appropriate list is searched for the point in the ordered list where the newly freed block should be reincorporated. If the new block is adjacent to the following free block, they are merged by

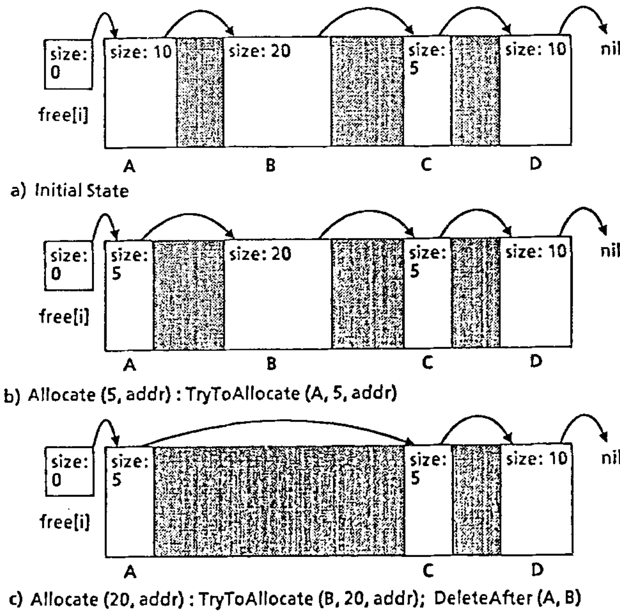

Fig. 2. Allocation requests.

deleting the following block (**DeleteAfter**) and continuing with a combined block. An attempt is made to append the new block on the end of the preceding block (**TryToAppend**). If that is unsuccessful, the new block is inserted as a separate node in the list (**InsertBetween**). Figure 3 illustrates a sequence of deallocation requests starting from the portion of the freelist given by Fig. 3a which shows the block X being released by the first call. In Fig. 3b, X has merged with the following free block, C. The procedures **DeleteAfter** and **InsertBetween** are involved in reaching this state. Figure 3b also shows the next block to be released, block Y. Figure 3c gives the state after appending Y onto block A (**TryToAppend**). Finally, block Z is to be freed. This block can not be merged with either of its neighboring free blocks and is inserted as a separate node in the freelist of Fig. 3d (**InsertBetween**).

Each of our parallel solutions entails the specification of the six procedures mentioned, namely **Allocate**, **Deallocate**, **TryToAllocate**, **TryToAppend**, **DeleteAfter**, and **InsertBetween**.

2.2. Monolithic Lock Approach

The simplest approach to memory allocation on a multiprocessor is to perform allocation and deallocation serially. This is the approach used in

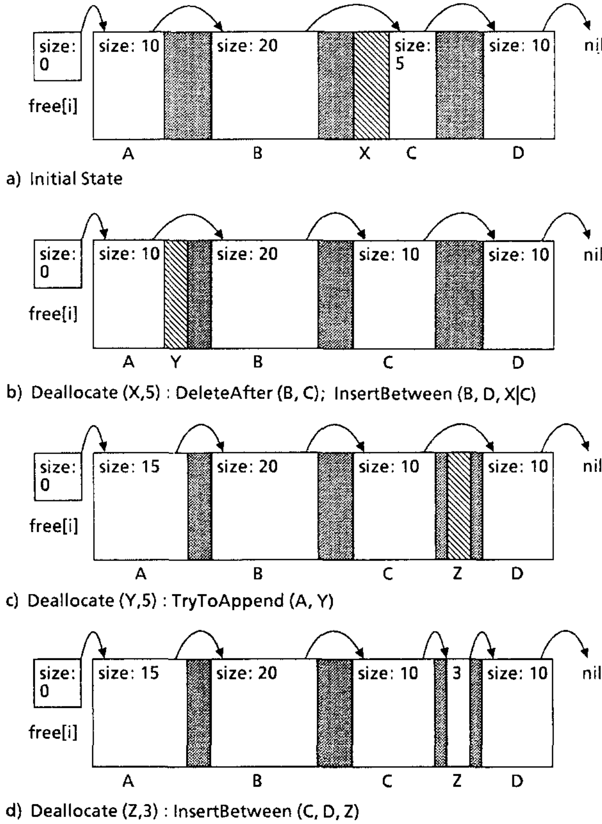

Fig. 3. Deallocation requests.

the original implementation of the Uniform System. In essence this is like placing a single exclusive lock on the freelist as a whole for the duration of the operation. Another interpretation is that the allocation routines are protected as entries into a monitor. This solution has been included primarily for comparison purposes and is referred to as algorithm 1. The changes to the sequential procedures are minimal, so we do not need to present pseudocode. Basically, the first statement in each of the procedures, **Allocate** and **Deallocate**, exclusively locks the freelist structure and the last action performed unlocks it.

An obvious variation on this scheme is to exclusively lock each individual sublist separately. The lock is placed after calculating the index of the list to be used. This approach (called algorithm 2) spreads out competition for locks. For allocation, the choice of list can be influenced by the length of waiting time for the lock at a particular candidate list. If the

number of processes making memory requests and the number of sublists from which memory may be allocated are equal, the expected number of users of a given sublist will be one and most lock requests will be granted immediately. This is the level of concurrency apparently called for in the Uniform System and it is essentially the approach that has been incorporated into recent releases.

2.3. Algorithms Based Upon Finer Granularity of Access

The two solutions presented in this section are based on techniques developed for concurrent access to database index structures such as B-Trees and hash tables. These techniques include specialized locking protocols, the use of atomic fetch-and-Φ operations, and structural modifications that reduce the need for locking. Locks are placed on individual nodes of the data structure. Thus, processes working in different regions of the same sublist (i.e. involving disjoint sets of nodes) can coexist without locking conflicts. The motivation behind these node granularity algorithms is to support higher degrees of concurrency where there may be many processes operating with a sublist.

Actually, these solutions go somewhat further than just reducing the granularity of locking and permit limited concurrent sharing of individual nodes. A common strategy in these solutions is to allow processes which are *searching* the freelist structure to use information in nodes concurrently accessed by other searching processes. Thus, efforts to find and reserve a large enough block of memory (in **Allocate**) or to locate the appropriate place in the list to reinsert a range of addresses (in **Deallocate**) are performed either with no locking or with compatible (read) locks. Not until there appears to be a need to change links between nodes does a process attempt to exclude other processes from the one or two nodes involved in structural modifications (i.e. by placing incompatible locks while inserting or deleting a node). The two solutions differ in the mechanisms used to ensure the structural integrity of the freelist.

The first node granularity algorithm (algorithm 3) uses two kinds of locks (read-locks and write-locks) and employs a technique that involves a particular pattern of lock requests and releases called *lock-coupling*. In a lock-coupling protocol, a process continues to hold a lock on one component of the structure until it acquires a lock on the next component (assuming some ordering) and then the previous component may be unlocked. The idea behind lock-coupling is that the presence of a process moving through the data structure can always be detected by other processes via incompatible lock requests. This property can be used to ensure that structural information needed by a searching process does not become

invalid or inaccessible as a result of a concurrent update. Thus, a process wanting to delete a node from the freelist (possibly because the entire block has been allocated) must request a write-lock on the predecessor of the node to be deleted and then on the node itself. This forces the removal of the node to be delayed until processes relying on old information (and holding read-locks) have moved on (releasing locks as they leave). The write-lock also blocks processes just arriving in this vicinity until the new structural state is in place.

The overhead of placing a read-lock at every node and the delays caused by the presence of incompatible locks are the costs incurred by searching processes in the lock-coupling approach. The final solution (algorithm 4) eliminates the need for lock-coupling by modifying the freelist data structure to provide temporarily valid paths through deleted list nodes. The key idea is to remove the header information from the allocatable memory space and maintain a separate linked list structure. This permits a block of memory to be released for use by the requesting process while temporarily retaining the header node so that its next pointer can provide a path back into the freelist.

Soution 4 avoids all locking overhead during the searching phase. Since there is no mechanism to prevent processes that hold pointers to a node which is being deleted from proceeding with the access, the node remains available and acts as a detour on the search path. However, in order to do modifications, structural information used must be correct. A pointer that is acquired without a lock can not be trusted and must be verified, once a lock is acquired, before structural modifications can be made. Deleted nodes are given a size field of zero to prevent allocations or adjacency tests from succeeding since they no longer represent a block of free space.

Garbage collection of deleted nodes is delayed until there are no processes that might possibly still need the structural information contained in them. The mechanism used to delay garbage collection ensures that none of the nodes present when a process begins a search pass through a sublist can disappear until it completes that pass, whether or not that process actually needs the nodes deleted in the meantime.

2.3.1. Intranode Operations

Both of these solutions use essentially the same procedures, **TryToAllocate** and **TryToAppend**, for changing the size of a node in the freelist. These operations use the *fetch-and-add* and *fetch-and-store* instructions applied to the size field describing the block to allow concurrent attempts to add or remove space in it. Thus, explicit locks are not needed

for concurrently reserving space or appending adjacent space onto the end of an existing free block.

The key idea underlying these algorithms is to atomically reserve some portion of the block by decreasing the size field using the fetch-and-Φ instructions. The **TryToAllocate** procedure reduces the size field by the amount of the request and is considered successful if it produced a nonnegative size value in that atomic step. **TryToAppend** atomically makes the size field negative to prevent concurrent operations upon the size value from being successful. This allows a nonnegative value returned from the atomic operation to be considered the stable and true size. Then, if the old block proves to be adjacent to the new block, the size of the combined block can be accurately calculated.

The one subtle point in the functioning of these routines is the restoration of the size field when the routine is unsuccessful. The possible values that the size field can assume are classified as *valid* and *invalid* values (defined slightly differently in the two solutions). The fetch-and-Φ operation may transform the contents of the size field from a valid value to an invalid value, from an invalid value to another invalid value, or from one valid value to another valid value (this last possibility occurs only in **TryToAllocate**). The unique process that made the valid to invalid transition among the set of all processes concurrently manipulating the size of a node has the sole right to restore the size field to the last valid value seen (i.e. the truth). This is done by a direct assignment. The key assertion used in arguing for the correctness of the manipulations on the size field is that, at any point in time, there is at most one process authorized to perform such an assignment on a node. This approach avoids some race conditions which could occur if each process that produced an invalid value were to apply an inverse operation to undo its own contribution to the invalid size.

2.3.1.1. Code and Discussion. The pseudocode procedures follow. They are presented in the form used by solution 3. The small adjustments made for solution 4 are described later. In this version, the valid size values are defined as "greater than zero."

```
        int TryToAllocate(B, request, N)
        node *B; /*pointer to freelist node*/
        int request;
0       node **N; /*pointer to allocated memory block*/
        {
          int oldsize;
```

```
      /*if it doesn't look promising, don't touch B->size*/
1     if (B->size < request) return FAIL;

      /* Atomically decrease B's size */
2     oldsize = FetchAndAdd(B->size, -request);

      /* If there was sufficient space, calc. starting address
      of region to be given to caller */
3     if (oldsize >= request) {
4       *N = B + (oldsize - request);
5       return SUCCESS;
6     }

      /* Insufficient space, we made the size invalid in trying,
      restore size */
7     if (0 < oldsize) B->size = oldsize;

8     return FAIL;
    }

    int TryToAppend(B, N)  /* N onto B */
    node *B, *N;
    {
      int oldsize;

      /* If apparently not adjacent, don't touch B's size */
1     if (B + B->size != N) return FAIL;

      /* Atomically decrease B's size */
2     oldsize = FetchAndStore(B->size, -1);

      /* If the two blocks were adjacent when we tried,
      merge them */
3     if (B + oldsize == N) {
4       B->size = oldsize + N->size;
5       return SUCCESS;
6     }
```

```
            /* if they weren't adjacent,
            we made the size invalid in trying, restore size */
7           if (0 < oldsize) B->size = oldsize;

8           return FAIL;
        }
```

After **TryToAllocate** reduces the size by the requested amount in line 2, testing the previous size value that is returned by the fetch-and-add instruction indicates whether or not there is sufficient space available (line 3). The pointer given to the caller in the case of a successful allocation is based on the old size returned from the fetch-and-add (line 4). It is possible for multiple concurrent requests to succeed within the same large block. If there is not enough space, the fetch-and-add results in a negative value being stored in the size field and the attempt fails. A **TryToAllocate** call that produces a negative value when the previous value was greater than zero is responsible for restoring the original size (line 7). Other concurrent processes that negate the size field still further (even those with smaller requests that could have been allocated space from this block) simply fail and rely on the first unsuccessful attempt to fix the size field since they have no information about the correct size. A **TryToAllocate** call that fails because of seeing an invalid size field may cause the request to search farther down the freelist than it would under sequential computation. The initial test (line 1) to see if the size of the block is greater than the request serves to avoid unnecessarily producing a negative value of size which could interfere with more promising concurrent requests. In all practical situations, this test represents a performance enhancement. However, without it, one can construct a pathological execution sequence of *only two* processes involving one process with a request too large to satisfy that denies another process with a small request from ever seeing free memory as the two processes proceed together in searching the freelist.

TryToAppend uses the fetch-and-store instruction to force the size field to take on a negative value (line 2). The first test for adjacency of blocks reduces interference with the size field by processes with poor prospects of successfully appending (line 1). Note that an execution sequence can be constructed such that more than one process executing **TryToAppend** can get past this point in the code. However, at most one process emerges from the fetch-and-store having seen a positive value for the old size and it either appends if the blocks are still adjacent (line 4) or it restores the old size if they are not adjacent (line 7). If the size field is negative when the fetch-and-store is executed, the **TryToAppend** call fails and an opportunity to

merge blocks may be missed. Thus, there is potentially greater fragmentation because of concurrency. However, this situation arises when there are parallel **TryToAllocate** calls (either by directly making the size value invalid or by enabling a second **TryToAppend** call to get past line 1). In usage patterns with distinct allocation and deallocation phases, this interaction is not a serious problem.

Because of the process interactions mentioned earlier, allocation algorithms using these routines are not strictly first fit and are not serializable in a traditional sense.

2.3.2. Algorithm with Read-locks and Write-locks

In this lock-coupling solution, two different lock types may be used so that processes can share a node by placing locks on it which are compatible. These locks are used for controlling interactions among processes that involve structural modifications of the freelist. The goal is for a process to hold the fewest locks and the least restrictive locks possible at each step of the operation.

The basic idea behind the **Allocate** and **Deallocate** procedures is to employ lock-coupling with read-locks from each node visited during the search of the freelist to its successor. The structure modifying procedures, **DeleteAfter** and **InsertBetween**, use write-locks for two purposes. Whenever the next pointer of a node is to be modified, the calling process must hold a write-lock on that node. The write-lock requested on a node being deleted is meant to ensure that other processes attempting to access that node have left before it is allowed to disappear (and the header information in it destroyed). Thus, the lock-coupled read-locking performed in **Allocate** and **Deallocate** interacts with the write-locking in **DeleteAfter** and **InsertBetween** to ensure that the effects of concurrent updates are seen.

The following table describes the compatibility relationships between lock types. When a lock request is made on a node of the freelist, the exist-

Table I.

Lock Request	Existing Locks	
	Read-Locks & no waiting Write-Locks	Write-Lock or waiting Write-Lock
Read-Lock	yes	wait
Write-Lock	wait	fail

ing locks held by other processes and whether there is a waiting write-lock request are inspected to determine if the new request will be granted (possibly after waiting for incompatible locks to clear) or will fail.

There are five operations available for manipulating locks: **ReadLock**, **ReadUnlock**, **WriteLock**, **WriteUnlock**, and **Purge & WriteLock**. The semantics of **ReadLock** and **ReadUnlock** are obvious from the compatibility table. The **WriteLock** procedure upgrades the calling process's existing read-lock to a write-lock when there are no other processes holding locks on the node and **WriteUnlock** downgrades the lock back to a read-lock. **Purge & WriteLock** preempts any write-lock request that is waiting for read-locks to be released and makes a new write-lock request on behalf of the calling process.

These locking semantics are more complex than those usually associated with read-locks and write-locks. A write-lock request that conflicts with an existing lock may wait (blocking the process) or fail (return without acquiring the lock). In addition, a waiting write-lock request may be converted into one that returns failure, even after a substantial time spent waiting. These semantics capture particular interactions between processes. Allowing only one process to wait on a write-lock request for read-locks to clear (causing other write-lock requests to fail) prevents a deadlock situation from developing when multiple processes, each already holding a read-lock, want to modify the same link. The preemption of waiting write-lock requests in **Purge & WriteLock** favors a more urgent request over a less important one.

2.3.2.1. Code and Discussion. The pseudocode and a more detailed discussion of the interactions among processes are given here.

```
          int Deallocate(N, size)
          node *N;
          int size;
          {
           node *F, *Fnext;
           unsigned searching;
           int i;

1          N->size = size; /*build header info within block*/
2          i = ChooseList(N);
```

```
3         while (TRUE) {
4           searching = TRUE;
5           F = &free[i];
6           ReadLock(F);
            /* Find the point to reinsert N into the free list */
7           while (searching && F->next != nil) {
8             Fnext = F->next;
9             if (Fnext > N) {
10              searching = FALSE;
                /* Attempt to merge N with the following block */
11              if (N + N->size == Fnext)
12                if (DeleteAfter(F, Fnext) == SUCCESS)
13                  N->size = N->size + Fnext->size;
14            }
15            else {
16              ReadLock(F->next);
17              ReadUnlock(F);
18              F = Fnext;
19            }
20          }
            /* Only F is read-locked at this point */
            /* Attempt to merge N with the preceding block */
21          if (TryToAppend(F, N) == SUCCESS) {
22            ReadUnlock(F);
23            return SUCCESS ;
24          }
            /* Insert N into the list */
25          if (InsertBetween(F, F->next, N) == SUCCESS) return SUCCESS;
26        }
        }
          int Allocate(size, N)
          int size;
          node **N;
          {
            node *F, *Fnext;
```

```
1       while (TRUE) {
2         F = &free[RandomIndex()];
3         ReadLock(F);

4         while (F->next != nil) {
5           Fnext = F->next;
6           if (TryToAllocate(Fnext, size, N) == SUCCESS){
              /* N is a proper suffix of F->next */
7             if (*N != Fnext) {
8               ReadUnlock(F);
9               return SUCCESS;
10            }
              /* If the entire F->next block was allocated,
              remove it from the free list */
11            if (DeleteAfter(F, Fnext) == SUCCESS){
12              ReadUnlock(F);
13              return SUCCESS;
14            }
              /* Undo the effect of TryToAllocate
              and go on down the freelist */
15            Fnext->size = size;
16          }
17          ReadLock(F->next);
18          ReadUnlock(F);
19          F = Fnext;
20        }
21        ReadUnlock(F);
22      }
      }
```

The **Allocate** procedure completes the picture of process interactions based on the size field of a block. In this solution, a size field of zero is interpreted as invalid by the **TryToAllocate** and **TryToAppend** routines. For example, a **TryToAppend** call that reads a zero in the fetch-and-store instruction does not find the blocks adjacent and does not restore the old size. A size of zero indicates that a process executing the **Allocate** procedure has reserved the entire block and is now responsible for the size value. The block is either being deleted or the size is being restored within **Allocate**

(line 15). In each of the three cases in which an assignment of a particular value is made to the size field (as opposed to a fetch-and-Φ), the existing size must be invalid and there is exactly one process capable of making the change (i.e. the one that saw the last valid value).

```
        int DeleteAfter(pred, del)
        node *pred, *del;
        {
1         if (WriteLock(pred) == FAIL)
2           return FAIL;

3         ReadLock(del);

4         pred->next = del->next;

5         WriteUnlock(pred);

6         Purge&WriteLock(del);
7         WriteUnlock(del);
8         ReadUnlock(del);

9         return SUCCESS;
        }
```

```
        int InsertBetween(pred, succ, ins)
        node *pres, *succ, *ins;
        {
1         if (WriteLock(pred) == FAIL) {
2           ReadUnlock(pred);
3           return FAIL;
4         }

5         ins->next  = succ;
6         pred->next = ins;

7         WriteUnlock(pred);
8         ReadUnlock(pred);
9         return SUCCESS;
        }
```

Consider the interaction between one process executing **DeleteAfter** (call it process *d*) and another process during its searching phase (process *s*). Let the node to be deleted be referred to as node *N* and its predecessor be node *P*. Imagine that *s* holds a read-lock on node *P* and has already read its next pointer which leads to *N*. The deleting process, *d*, must write-lock *P* before modifying its next pointer and making the value that *s* saw obsolete. The granting of this lock request is prevented by the read-lock held by *s*. That lock is not released until *s* has either completed its requested operation using *N*, failed trying to write-lock *P* while *d*'s lock request is waiting, or acquired a read-lock on *N* to continue its search. Thus, the pointer remains valid as long as it is needed. Similarly, *d* places a read-lock on *N* to protect the validity of its next field while modifying *P*'s next field to point to *N*'s successor. At this point *N* can only be seen by processes that acquired the pointer to it and locked it before it was removed. Finally, *d* acquires and then immediately releases a write-lock on *N* before returning from **DeleteAfter** to ensure that all processes relying on *N*'s contents and holding read-locks on it have moved on. This gives *d* exclusive access to *N*.

The interaction between a searching process and a process executing **InsertBetween** is relatively simple. As in the previous case, the inserting process must acquire a write-lock on the predecessor node before changing its next pointer and this conflicts with read-locks held by searching processes depending on this value.

Structural modifications involving the same node in the list (i.e. as an argument to **DeleteAfter** or **InsertBetween**) are applied serially because of the locking protocol. Consider the case of two processes both trying to insert a block at the same place between *P*, the intended predecessor node, and *N*, the successor. Both of these processes already hold a read-lock on *P* and now make a request for a write-lock on *P*. Assuming no other processes in the vicinity of *P*, one of these processes waits for the write-lock because of the other's read-lock. The second write-lock request (by the other process) fails and causes the insertion attempt to be aborted, releasing the read-lock. This prevents a deadlock situation and avoids the need to reevaluate the proper placement of the block to be inserted once a write-lock has been acquired. Two attempts to delete the same node can arise because of one process doing an allocation of the exactly the size of the block in question and another process deallocating and trying to merge an adjacent block. As before, the second write-lock request on the predecessor fails. Similar arguments hold for mixing processes making concurrent calls to **InsertBetween** and **DeleteAfter** at node *P*. At most one write-lock request succeeds.

Interactions involving neighboring nodes are also interesting. Consider

three directly linked nodes in the freelist, *A*, *B*, and *C*, such that the next pointer of node *A* leads to *B* and the next pointer of *B* leads to *C*. There are two processes, *d* and *u*. Process *d* is trying to delete node *B* by executing **DeleteAfter(A,B)**. It holds a read-lock on *A* prior to the call. Imagine that *d* has successfully gained the write-lock on *A*, the read-lock on *B*, and has changed *A*'s next pointer, making *B* unreachable through the freelist. Process *u* is trying to perform some update after node *B* (i.e. either **DeleteAfter(B,C)** or **InsertBetween(B,C,X)**). It holds a read-lock and has requested a write-lock on *B* which is waiting on *d*'s read-lock. Process *d* now needs to momentarily write-lock node *B* to ensure that there are no processes still requiring its header information. Unfortunately, there is already a waiting write-lock on *B*. Since the deletion operation is in a sense committed to finishing, preemption is used to prevent deadlock (i.e. the use of the **Purge & WriteLock** operation). Process *u*'s write-lock request that was waiting now fails, eventually leading to the release of the read-lock on *B*.

2.3.3. Algorithm with Separate Header Lists

The lock-coupling of the previous solution is used to ensure that structural information needed by a searching process does not become invalid or inaccessible as a result of a concurrent update. An alternative, which eliminates the locking overhead experienced by searching processes at every node, is to permit a degradation in the quality of the list structure seen by processes. Processes can be allowed to follow obsolete pointers if deleted nodes remain available and provide information to help such "lost" processes find a path back into the freelist. A problem with retaining deleted nodes when they reside within the free memory blocks themselves is that a fully allocated block can not be immediately used by the requesting process. This suggests the strategy of separating the header information from the allocatable memory space. A portion of memory is now reserved for headers and not available for allocation. As a side effect, smaller allocation units are now possible since blocks do not have to be large enough to accommodate all of the header information. The new problem of allocating and deallocating header nodes is much simpler than the original allocation problem because they are of a fixed size and not constrained by address ordering. Figure 4 shows the freelist structure with the separate header space. Each header node includes an address field containing a pointer to the starting location of the free memory block. A header node that has been deleted from the freelist (i.e. it represented a block that has been either allocated or merged into another block) but is still available to searching processes is given a size field of zero. This

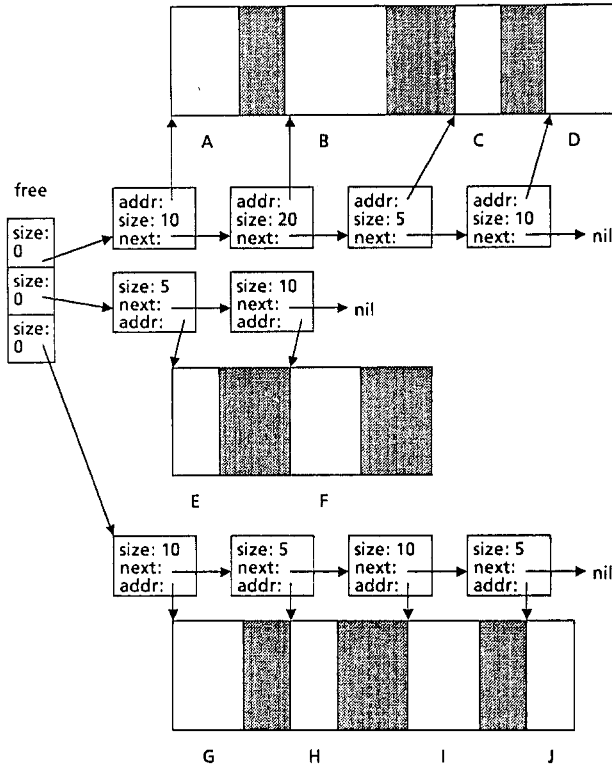

Fig. 4. Freelist with separate headers.

prevents allocations (in **TryToAllocate**) and adjacency tests (in **TryTo-Append**) from succeeding. Zero is now considered a valid size. The previous size is restored on unsuccessful **TryToAllocate** and **TryToAppend** calls if it was greater than *or equal to* zero and the new value is negative. All of the memory in a block can be allocated even when the attempt to delete the header node does not succeed. Thus, nodes with a size field of zero can be left in the actual freelist as well as among the deleted, but still accessible, nodes.

This solution uses two different kinds of locking. The first type of lock is a write-lock placed on individual header nodes. These write-locks are used to control interference among the structure modifying operations. In this solution, a process places no locks on individual header nodes until it tries to change the structure of the sublist it has chosen to search. A searching process does not use incompatible read-locks to detect the presence of another process performing **DeleteAfter** or **InsertBetween** in its

immediate vicinity. The data structure can change between the time a searching process reads a next pointer in a node and the time it acts upon this information by accessing the indicated node. A newly inserted node may be skipped or the destination may already be deleted by the time the process arrives. Structural changes, however, must be based on an accurate view of the current state. This means that a process attempting to modify the structure must reassess what the current structural relationships actually are after acquiring its write-locks.

The operations available for manipulating write-locks are **WriteLock** and **WriteUnlock**. Calls to **WriteLock** can fail for one of two reasons: the node may be already write-locked by some other process (i.e. write-locks are incompatible with other write-locks) or it may have been deleted. An attempt to place a write-lock on a node implies that the node is to be involved in some structural change which is obviously inappropriate for a node that has already been deleted from the list.

The second type of locking actually represents a mechanism for triggering garbage collection of deleted header nodes. This mechanism may be implemented by reference counts associated with certain components of the sublist structure; however, locking terminology captures the interac-

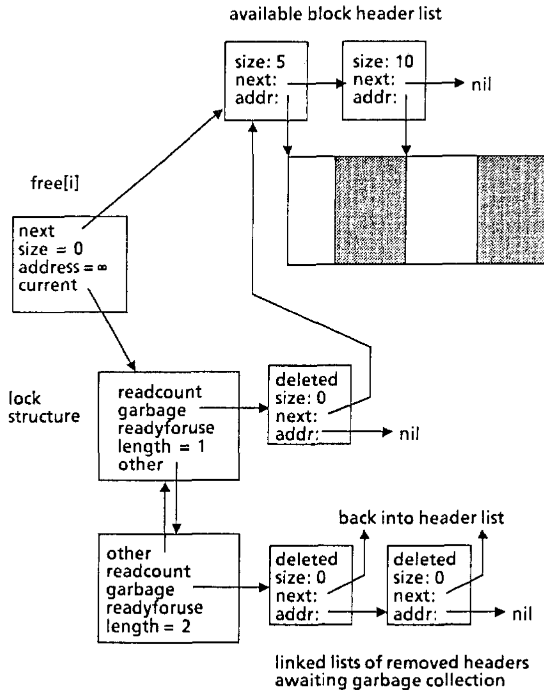

Fig. 5. Details of versions.

tions in a concise way. Thus, we define two lock types that are employed at the granularity of a whole sublist: self-compatible read-locks (i.e. totally compatible with the read-locks of other processes) and exclusive GC-locks. These locks are used to protect a *version* of the sublist from garbage collection. Specifically, none of the header nodes present when a process acquires its read-lock on the sublist can disappear until after that lock is released.

These versions are represented by lists of nodes that have been removed from the freelist and are waiting to be destroyed. Two lists are sufficient. One represents the current version which consists of very recently deleted nodes (i.e. those deleted from the freelist structure since this list was designated current). Read-lock requests are made to the current version. Thus, each process is associated with the version which was current at the time it acquired its read-lock. The other (passive) list contains nodes that will be freed by a background **GarbageCollector** process as soon as all processes holding read-locks on the old version release them (i.e. when a request to place a GC-lock can be granted). Once the deleted nodes on the passive garbage list have been reclaimed, it is again possible to swap the roles of two lists. This approach is similar to the garbage collection mechanisms in Refs. 18 and 19. Figure 5 shows the data structure associated with versions, including the lists of header nodes awaiting garbage collection.

Note that the write-locking protocol is distinct from the read-lock/GC-lock scheme. They are locking different kinds of things in the data structure and there are no compatibility constraints between them.

2.3.3.1. Code and Discussion. The procedures, **TryToAllocate** and **TryToAppend**, must be modified slightly for this new freelist structure. In particular, the addresses of the freelist nodes and the memory blocks they describe are no longer the same and address calculations are based on the address field of the header node. The substitutions for the designated lines in the previous code bodies are given here:

```
(line 0 in TryToAllocate)
block **N; /*pointer to memory block*/

(line 4 in TryToAllocate)
*N = B->address + (oldsize - request);

(line 7 in both TryToAllocate and TryToAppend)
if (0 <= oldsize) B->size = oldsize;

(line 1 in TryToAppend)
if (B->address + B->size != N->address) return FAIL;
```

```
(line 3 in TryToAppend)
if ((oldsize!=0) && (B->address + oldsize == N->address)) {
```

There are three operations available for manipulating the two lock types associated with garbage collection: **ReadLock**, **ReadUnlock**, and **GCLock**. The call to **ReadLock** returns a pointer to the current version. This number is supplied subsequently as an argument in the matching **ReadUnlock** call. **ReadLock** always succeeds since it is directed at the current version and there are no incompatible locks held on this version. It essentially increments a reference count (free[i].current → readcount) within a critical section. **GCLock** is directed at the passive version and its primary function is to wait until all the read-locks held on this version are released. No new read-lock requests are made for a version that could have an outstanding GC-lock request. Therefore, there is no need to *hold* a GC-lock and so, no need to explicitly release one. The **GCLock** procedure essentially waits until the reference count (free[i].current → other → readcount) is zero.

For the moment, we will ignore the issue of garbage collection and assume that deleted header nodes are always available if a process attempts to access one. The pseudocode procedure bodies for **Allocate** and **Deallocate** are given here. **ReadLock** and **ReadUnlock** calls are the only evidence of garbage collection appearing in these routines.

```
        int Deallocate(n, size)
        block *n;
        int size;
        {
          node *F, *N;
          int i;
          version *myversion;

1         i = ChooseList(n);

          /*alloc and init a new header for block n*/
2         N = GetHeader(i);

3         N->size = size;
4         N->address = n;

5         while (TRUE) {
```

```
6           F = &free[i];
7           ReadLock(F, &myversion);
8           if (DeallocPass(N, F, myversion, i) == SUCCESS)
9             return SUCCESS;
10        }
        }

        int DeallocPass(N, F, myversion, i)
        int  i;
        node *F, *N;
        version *myversion;
        {
          unsigned searching, locked;
          node *Fnext;
          int oldsize;


1         searching = TRUE;
2         locked = FALSE;

          /* Find the point to reinsert N into the free list */
3         while (searching && (Fnext = F->next != nil)) {
4           if (Fnext->address > N->address)
5               searching = FALSE;
6           else F = Fnext;
7         }

8         Fnext = F->next;
9         if (Fnext != nil) {
            /* Attempt to merge N with the following block */
10          if (N->address + N->size == Fnext->address)
11            if (WriteLock(F) != FAIL) {
12              locked = TRUE;
                /* now that F is locked, is Fnext still the
                successor?*/
13              if (F->next == Fnext)
```

```
14              if (DeleteAfter(F, Fnext, i, &oldsize)
                === SUCCESS)
15                N->size = N->size + oldsize;
16          }
17      }

        /* Attempt to merge N with the preceding block */
18      if (TryToAppend(F, N) === SUCCESS) {
          /*enqueue N on garbage list*/
19        remove(N, i);
20        if (locked) WriteUnlock (F);
21        ReadUnlock(myversion, i);
22        return SUCCESS ;
23      }

24      if (locked === FALSE)
25        if (WriteLock(F) === FAIL) {
26          ReadUnlock (myversion, i);
27          return FAIL;
28        }
        /*F is now locked, so F->next is stable*/
29      Fnext = F->next;

        /*Did an insertion occur?*/
30      if ((Fnext != nil) &&
        (Fnext->address < N->address)){
31        WriteUnlock(F);
32        ReadUnlock(myversion, i);
33        return FAIL;
34      }

        /* Insert N into the list */
35      InsertBetween(F, Fnext, N, i, myversion);
36      return SUCCESS;
```

```
        int Allocate(size, n)
        int size;
        block **n;
        {
          node *F, *Fnext;
          version  *myversion;
          int i, oldsize;
1         while (TRUE) {
2           F = &free[i = RandomIndex()];
3           ReadLock(F, &myversion);

4           while (Fnext = F->next != nil) {
5             if (TryToAllocate(Fnext, size, n) == SUCCESS){
                /* n is a proper suffix of block in F->next */
6               if (*n != Fnext->address) {
7                 ReadUnlock(myversion, i);
8                 return SUCCESS;
9               }
                /* If the entire F->next block was allocated,
10              remove it from the free list */
11              if (WriteLock(F) != FAIL){
                  /*F is locked- is Fnext still the
                  successor?*/
12                if (Fnext == F->next)
13                  DeleteAfter(F, Fnext, i, &oldsize);
14                WriteUnlock(F);
15              }
16              ReadUnlock(myversion, i);
                /*if attempt to delete fails, leave Fnext
                in list with zero size*/
17              return SUCCESS;
18            }
19            F = Fnext;
20          }
21          ReadUnlock(myversion, i);
22        }
```

Aside from the changes made necessary by the modified data structure, the fundamental way in which this solution differs from the previous one is in the inability of a process to trust the structural information it reads from a node until it has placed a write-lock on that node. In particular, it is necessary to be sure that Fnext (the node to be deleted) is still the direct successor of F before calling **DeleteAfter** (line 13 in **DeallocPass** and line 12 in **Allocate**). Another example is the need to make sure that the place between F and $F \rightarrow$ next is still appropriate for inserting a header node describing the block at address n (line 30 in **DeallocPass**). A concurrent **InsertBetween** call may have inserted a node following F for a block with an address less than n. Because of this need for this extra checking after the write-lock is in place, the **WriteLock** calls on F have been moved out of **DeleteAfter** and **InsertBetween**. If a write-lock is acquired in an attempt to merge with the following block (line 11 in **DeallocPass**), it is held for the possible subsequent call to **InsertBetween** in order to eliminate the potential for more interference. This explains the purpose of the flag, locked, which indicates whether a write-lock has already been granted or not. If the **DeleteAfter** call is successful (line 14 in **DeallocPass**), this call to **DeallocPass** is guaranteed to succeed. In particular, the condition in line 30 can never be satisfied in that case. No write-locks are required for the attempt to merge with the preceding block (lines 18–23 in **DeallocPass**) because the second adjacency test in **TryToAppend** (line 3—after size has been made invalid) accurately captures the relevant structure even if it has been undergoing changes.

Write-locks are held during executions of **DeleteAfter** and **InsertBetween** to prevent direct interference by another update and to check whether the node is garbage. Since searching processes may be concurrently performing fetch-and-add instructions on the size field of a node being deleted, the process executing **DeleteAfter** must ensure it sees a true size. This is because the value is returned from **DeleteAfter** to be used in case the node is being merged. The process first waits for a valid size and then invalidates it using a fetch-and-store to get the size value and prevent additional interference. Then the size field of the deleted node must be set to zero so that **TryToAllocate** and **TryToAppend** calls do not succeed. The pseudocode procedures follow:

```
        int DeleteAfter(pred, del, i, oldsize)
        node *pred, *del;
        int i, *oldsize;
        {
1         if (WriteLock(del) == FAIL)
```

```
2           return FAIL;

3         pred->next = del->next;

          /*loop until del->size is valid
           and leave with del->size < zero, thus reserving it*/
4         while (*oldsize =
          FetchAndStore(del->size, -1) < 0)
5           while (del->size < 0) delay;
          /*make it exactly zero*/
6         del->size = 0;
7         del->deleted = TRUE;

          /*enqueue del on garbage list*/
8         remove (del, i);

9         WriteUnlock(del);
10        return SUCCESS;
        }

        int InsertBetween(pred, succ, ins, i, myversion)
        node *pres, *succ, *ins;
        int i;
        version *myversion;
        {
1         ins->next  = succ;
2         pred->next = ins;

3         WriteUnlock(pred);
4         ReadUnlock(myversion, i);
        }
```

The presence of deleted header nodes motivates another change from the previous solution. In the **Allocate** routine of algorithm 3, if it is not possible to delete a block that was entirely allocated, the storage is essentially given back and the search continues down the freelist. After all, the storage locations occupied by the header information must be maintained

and can not be used by the caller if the block can not be removed from the freelist. Thus, there are no nodes with size zero (invalid) left in the linked list, lengthening search paths, in algorithm 3. In this solution, however, those nodes may be left in the freelist if deleting them fails for any reason (lines 11–17 in **Allocate**) and the caller is free to use the block of storage that the node described. As a result, the allocation request does not have to pass up eligible blocks of storage and zero is interpreted as a valid size value. There is also a correctness consideration that argues for allowing nodes with size of zero. Deleted nodes also contain a (valid) size field of zero that is set in **DeleteAfter** after obtaining a valid value (lines 4–6). If **Allocate** were to restore the size field as it does in algorithm 3, it would introduce the possibility of two processes both able to assign a value to the size field (e.g. a **DeleteAfter** call due to a merge attempt and a failed **DeleteAfter** called from **Allocate**). This can lead to a race condition and an incorrect size value. The statement of the assertion formulated for the last solution concerning assignments to the size field still holds in this solution.

To understand this solution, it is necessary to appreciate the role played by deleted nodes. First consider a process in the search phase of **Allocate** (process s) executing concurrently with another process trying to delete node N (process d). After process d executes line 3 in **DeleteAfter**, node N is no longer accessible from the freelist, but its next pointer is part of a path back into the freelist. Let node P be N's old predecessor in the freelist. Usually, $N \rightarrow \text{next} = P \rightarrow \text{next}$ at the time another process follows the deleted node's next pointer. However, multiple calls to **DeleteAfter** can yield longer paths before a freelist node is reached. If process s does the fetch-and-add (line 2 in **TryToAllocate**) on N before process d has reached line 4 in **DeleteAfter**, the allocation can still succeed in spite of the fact that the node is being deleted. If the **TryToAllocate** fails (possibly because process d has executed lines 4–6), process s follows N's next pointer and the search can continue.

Next, let proces s be in the search phase of **Deallocate** in parallel with process d. Now, the point at which the detour through deleted nodes rejoins the freelist matters significantly more. In particular, nodes inserted immediately following N's predecessor since the deletion of N will be missed. This may cause the current pass through the list to fail if the appropriate place to reincorporate the deallocated storage is in this unseen part of the freelist. Suppose node N has been deleted since process s decided to access it (line 3 in **DeallocPass**). Assume that the node, $N \rightarrow \text{next}$, is a legitimate member of the freelist structure. Process s determines that the address of the block of storage it is in charge of deallocating is less than the address represented by $N \rightarrow \text{next}$ (line 4 in the next iteration of loop 3–7 in **DeallocPass**). If process s tries to merge with $N \rightarrow \text{next}$, the

lock request on N will fail since it has been deleted or is still locked by process d. If process d is past line 6 in **DeleteAfter** or the **DeleteAfter** call is a result of complete allocation, a **TryToAppend** call by process s fails. If node N is being deleted to merge with another block and process d has not reached line 6, a **TryToAppend** can succeed. This means a bigger block (size of node N plus process s's block) will be returned from **DeleteAfter** to be merged with process d's block. If it appears necessary for process s to call **InsertBetween**, that call will never be made because the prerequisite write-lock request will fail. This strategy forces a reevaluation of the appropriate place to put the newly released storage when it falls in the gap between N's old predecessor and N's successor.

Now we turn to the problem of recycling deleted header nodes. A header must remain available as long as any process that may have read a pointer to it is still active. The read-lock placed on the version indicates the existence of a process still requiring that version. When the last read-lock on the old version is released by a call to **ReadUnlock**, garbage collection on the garbage list associated with that version can begin. There is a background **GarbageCollector** process associated with each sublist. The **GarbageCollector** periodically moves nodes from the garbage list to the list of available header nodes and marks the version as ready to become the current one again. The **Remove** routine enqueues deleted header nodes on the current garbage list and swaps the versions when the length of the current list exceeds some threshold and the other version is marked as ready. Pseudocode is presented here:

```
         GarbageCollector (i,myversion)
         int i;
         version *myversion;
         /*i and myversion are process creation parameters
         initially myversion = initial value of
         free[i].current*/
         {
1          while(TRUE){
2            Sleep(interval);
3            if (myversion != free[i].current){
4                GCLock(myversion);
5                Append (myversion->garbage, freeHeaders);
6                myversion->readyforuse = TRUE;
7                myversion = myversion->other;
8            }
```

```
9         }
        }

        Remove (N, i)
        node *N;
        int i;
        {
          version *ver;

1         ver == free[i].current;
2         Enqueue(N, ver->garbage);
          /*atomically*/
3         FetchAndAdd (ver->length, +1);
4         if (ver->length >= THRESHOLD) {
5           P(Mutex);
            /* Provides mutual exclusion with the
            reading of current version in ReadLock
            routine */
6             if ((ver == free[i].current) &&
7             (ver->other->readyforuse == TRUE)){
8               ver->readyforuse = FALSE;
9               free[i].current = ver->other;
10            }
11          V(Mutex);
12        }
        }
```

As a process begins each pass through one of the sublists of the freelist, it places a read-lock on the current version of that sublist (line 7 in **Deallocate**, line 3 in **Allocate**). The lock is held until either the operation succeeds (lines 7 and 16 in **Allocate**, line 21 in **DeallocPass**, and line 4 in **InsertBetween**) or the present pass is abandoned (line 21 in **Allocate**, lines 26 and 32 in **DeallocPass**). The designation of which version is current may change once during any particular pass. The end of each pass presents an opportunity to lock a potentially different version in the next pass, should one be necessary.

The **GarbageCollector** process periodically wakes up and checks whether the version it plans to collect next (myversion) has become the old

version (line 3). If so, it requests the GCLock to wait for read-locks to clear. Then, it moves deleted header nodes off the garbage list associated with the passive version and, once that list is empty, marks it as ready to reassume the role of current version (line 6). The current version designation can switch as soon as the old garbage list has been cleared (lines 7–9 in **Remove**). Finally, the **GarbageCollector** turns its attention to the other version, making it the candidate for its next round of garbage collection operations. Thus, its private version moves back and forth, following free[i].current.

The first task of the **Remove** routine is to insert the header node supplied as an argument onto the current version's garbage list. To be more precise, the process executing **Remove** initially reads which version is current, remembering it in a private variable, ver (line 1). Then, it directs all subsequent manipulations to that list even though it may not continue to be the current one. Note that this is not necessarily the version on which the caller holds its read-lock because another process calling **Remove** may have switched versions since that read-locked was placed. This means that the deleted node may have a longer lifetime than actually needed if it survives through the next version as well as the version that the deleting process locked. In this solution, the actual job of switching versions is done by a process executing **Remove** when what it believes to be the current garbage list has grown sufficiently long. Multiple processes executing **Remove** concurrently may all detect the length of the garbage list exceeding the threshold. Therefore, processes must enter a critical section to accomplish the actual switch. Once in the critical section it is necessary for the process to determine whether some other process may have beaten it to the act of doing the switch or whether the other list is not yet ready to become current (lines 6 and 7 in **Remove**).

3. EXPERIMENTS

The solutions being compared in our experiments are based on essentially the same underlying approach (the first fit allocation algorithm) and very similar freelist structures (i.e. both structures are linked lists ordered by the starting addresses of free blocks). Thus, the differences among the solutions are primarily related to the granularities of locking and memory access and the synchronization protocols used. The experiments are intented to capture the effects on performance of these kinds of design decisions. They also provide some insight into the impact that subtle architectural differences in memory architecture can have on the behavior of these kinds of shared memory algorithms.

The algorithms under consideration have been fully implemented to

perform the allocation and deallocation functions as outlined. Thus, our experiments measure the performance of the implemented operations rather than model the behavior of the algorithms more abstractly. This approach ensures that no details that may contribute to performance are overlooked, but it also complicates the evaluation by involving the idiosyncrasies of a particular parallel machine. The implementation of algorithm 4 includes the garbage collection mechanism for header nodes. The measurements taken for the requested operations in algorithm 4 have been appropriately adjusted to reflect the overhead due to garbage collection. Actually, one concession has been made to the fact that the development of these programs was intended for experimentation rather than actual use: algorithms 1 and 2 have been merged into a single implementation since, in the design of our experiments described here, they are essentially the same.

A driver program, running under the Uniform System, simulates a workload of multiple processes making requests to a system with a specified number of separate memories (i.e. sublists of the freelist). In all of our experiments, the number of memories specified has been one. This allows us to generate processor/memory ratios of up to 64/1. Actually, only 60/1 on the 64-node Butterfly Plus since some nodes are tied up by the system. Sixty processors making parallel memory allocation requests on a single memory sublist is an intense workload that seems very unlikely in a NUMA machine such as the Butterfly. One expects ratios closer to 1 in a machine with a memory module residing at each node. However, this strategy tests the range of applicability and scalability of the various solutions. Given the decision to use only one sublist in our experiments, the justification for only one implementation for both algorithms 1 and 2 is clearer. The results obtained can be interpreted appropriately for either algorithm. In the case of algorithm 1, the level of parallel demand is considered to be system-wide; whereas in algorithm 2, it is only applied to one component of a system that could have other sublists experiencing similar workloads.

The driver has three distinct phases in which requests are issued for allocation or deallocation of memory. After initially building the freelist structure and the chunks of allocatable memory, it generates tasks that make allocation requests in parallel until a target memory utilization has been reached. In this startup phase, allocations compete only with each other. Then during the steady state phase, each process generates a relatively balanced request pattern with allocations and deallocations distributed to keep memory utilization close to the specified target figure. Finally, the memory is released in the windup phase. Deallocation requests compete among themselves for access to the freelist in this phase. Each phase is measured separately.

The basic structure of tasks is a loop that generates the next request and makes the appropriate call. The choice of operation depends on the phase and (in steady state) on the current local view of memory utilization. For an allocation, the size of the request is selected randomly from a given distribution. For deallocation, a block is taken off the local list of blocks that have been previously allocated to this process. The loop terminates when the target allocation is reached in startup, after a specified length of time in steady state, and when all memory is returned in windup.

The parameters for each test include the number of processes making requests, the number of separate memories (always 1 in this set of tests), the duration of the steady state phase, the pattern of sleep times between requests by a process, and the target memory utilization. Although the capability for a random sleep interval has been provided, the results described here are based on a stressful workload in which the processes do not sleep between subsequent requests. Consequently, they are oriented toward the peak load situation.

Our primary measures of performance are throughput, average execution times for individual operations, the length of search paths in allocation requests, and the number of missed opportunities for merging adjacent free blocks observed in snapshots of the freelist. The throughput measure reflects how much parallelism can be achieved and how much overhead is inherent in each solution. The operation times show the costs of various sources of contention and the complexity in the algorithms. The search path measure is an indication of both fragmentation and interference among processes in the **TryToAllocate** routine. The search paths get longer as processes pass up acceptable blocks that just *appear* too small because the size fields have been reduced. Ideally, one might like a count of aborted merger attempts that have been caused by interference among processes and that should have succeeded. Unfortunately, this information can not effectively be gathered. We have settled for taking a snapshot of the freelist and counting adjacent blocks that have not been merged and recognize that it represents an underestimate of the fragmentation caused by parallel access.[5]

We first present the results for the steady state phase. The throughput curves for each of the three solutions running on the Butterfly Plus multiprocessor are shown in Fig. 6. Similar results on the Butterfly One are

[5] Unmerged blocks that exist at one point in time may subsequently get allocated again and successfully merged in a later deallocation. Evidence of the first failed merge attempt is lost. Note that although it is possible to have other processes look for unmerged blocks during normal searches and make up for any failed merge attempts they find, this has not been done in these implementations. Other processes do not routinely merge blocks they are not responsible for.

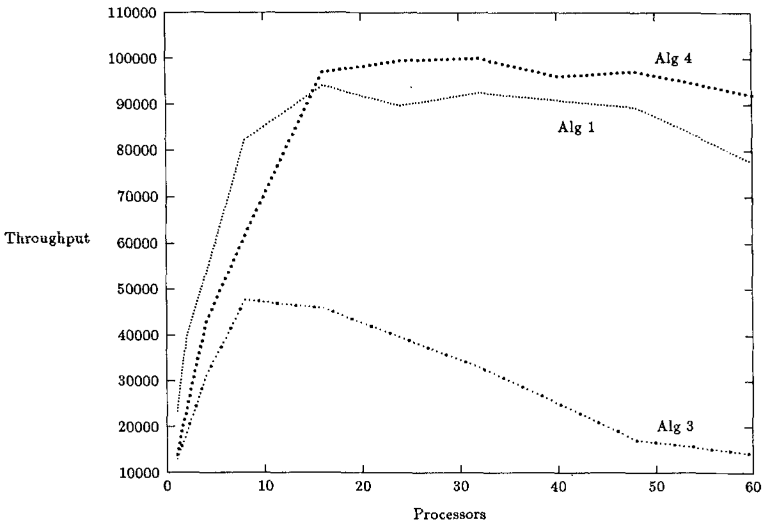

Fig. 6. Steady state throughput (on Butterfly plus).

presented in Fig. 7. The throughput is given as the number of allocate (deallocate) operations executed during the fixed time of the steady state phase (50 seconds) for various processor configurations.

One obvious conclusion that can be drawn from these graphs is that algorithm 3 is not competitive with the alternatives at any number of

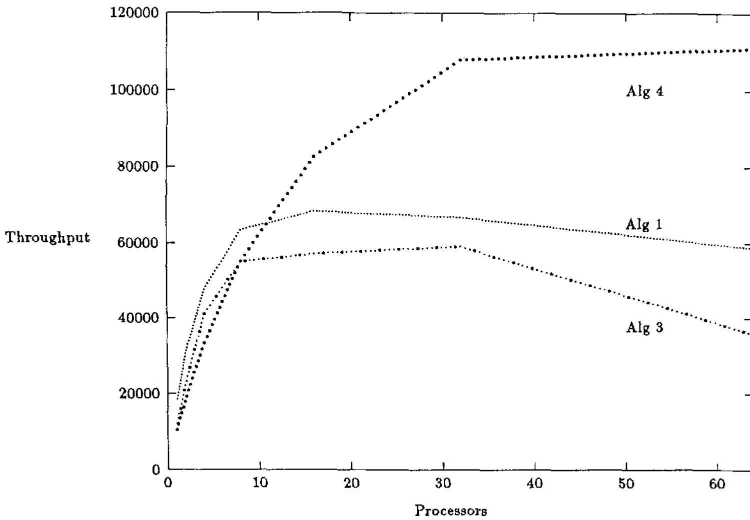

Fig. 7. Steady state throughput (on Butterfly one).

processors. The locking overhead of algorithm 3 appears to be quite significant. This statement refers to the work involved in handling lock requests at each block even when they can be satisfied immediately. The complex lock semantics certainly contribute to this overhead.[6] This high level of overhead (before considering interference by other processors) means that algorithm 3 would have to be very successful in delivering a high degree of parallel access in order to overcome this cost. Lock contention and contention for the memory module where the freelist resides limit the parallelism and slow execution down still further. Therefore, we do not need to seriously consider algorithm 3 in the rest of the discussion of steady state behavior.

The general shape of the throughput curves for algorithm 1 (algorithm 2 on 1 sublist) and algorithm 4 agree in Figs. 6 and 7, although the magnitude of the values and the crossover points are different. On small numbers of processors (i.e. less than 16), algorithm 1 delivers higher throughput because it is the simpler algorithm. It does not permit parallel access to the freelist, although parallel tasks executing algorithm 1 can overlap generation of the next request. Algorithm 4 has not achieved enough parallelism at this point to make up for its higher cost. After the crossover point, algorithm 4 is relying on parallelism to deliver the higher throughput. Note that the throughput curve levels off fairly rapidly (and in the Butterfly Plus appears to be starting down) with increasing numbers of processors. The write-locking protocol used primarily within deallocation operations is one factor in the performance of algorithm 4. However, there is also a (short) critical section within the implementation of the read-lock needed to determine the current version and in the **Remove** routine to update which version is current. These critical sections and increasing memory module contention appear to be contributing significantly to the limited parallelism.

A comparison of the throughput results on the two machines is extremely interesting. The redesign of the memory architecture in the Butterfly Plus is supposed to provide faster access to both local and remote memory. The increase in throughput exhibited by algorithm 1 in moving from the Butterfly to the Butterfly Plus is consistent with an overall faster memory access. However, the performance of algorithm 4 degrades in the transition between machines. The explanation seems to be that memory

[6] This is confirmed by tests of a simpler node granularity algorithm which performs lock-coupling with simple exclusive spin-locks. The less expensive locks produce somewhat better performance than algorithm 3 on small numbers of processors (although still not competitive with algorithm 1). The simpler algorithm suffers from restricted parallelism at higher numbers of processors, with throughput falling below that of algorithm 3.

module contention begins to exert its effects much earlier (at fewer numbers of processors) in the Butterfly Plus. The gap between local and remote access times has widened in the redesign.[7] Thus, the NUMA character of the Butterfly Plus is much more significant than it was in the original Butterfly design. Increasing the proportion of program execution that is spent on remote access puts greater pressure on shared memory. The fine granularity access to the freelist which is the basis for algorithm 4 makes contention an inevitable problem at some point. It appears that this point has shifted significantly between the two designs. Algorithm 4 is the clear winner in the results from the original Butterfly where local and remote accesses are more comparable. From this, one might conclude that fine granularity algorithms developed to provide high levels of concurrency, such as algorithm 4, are useful over a relatively wide range of configurations when the memory access times are fairly uniform. Depending on the ratio of remote to local access times in a NUMA design, the range of processor configurations between the point at which parallel access starts paying off and the point at which memory contention takes over may be limited for this class of algorithms.

The search paths followed in trying to satisfy an allocation request using algorithm 4 are 25–66% longer than those encountered using algorithm 1 (when there are $\geqslant 16$ processors). The longer search paths contribute to the cost of an individual operation. It appears that these longer paths are primarily caused by passing over blocks that are suitable but have an invalid size value. Another explanation is that the freelist structure is more fragmented because of missed opportunities to merge small blocks into larger ones. However, snapshots of the freelist structure taken at the end of the steady state phase show that algorithm 4 leaves (on the average, per run) about 1 pair of adjacent blocks that should have been merged and were not. The maximum number observed has been only 5 unmerged blocks that survived to the end of the steady state phase.

Figures 8 and 9 give the average time for an allocation and a deallocation, respectively, on the Butterfly Plus. The difference in the basic costs of algorithms 1 and 4 is evident in the values at one processor. The growth in the average operation cost of algorithm 1 is due primarily to lock wait time. The costs incurred by algorithm 4 have already been mentioned.

The results from the startup and windup phases give an indication of how the algorithms behave in a 2-phase request pattern (where the application has a separate allocation and deallocation phase). Results presented here are those from experiments on the Butterfly Plus.

[7] More improvement has been made in local access time than in remote access time. The access path to local memory has been made more direct and there is evidence that local accesses are favored in contention with remote accesses.

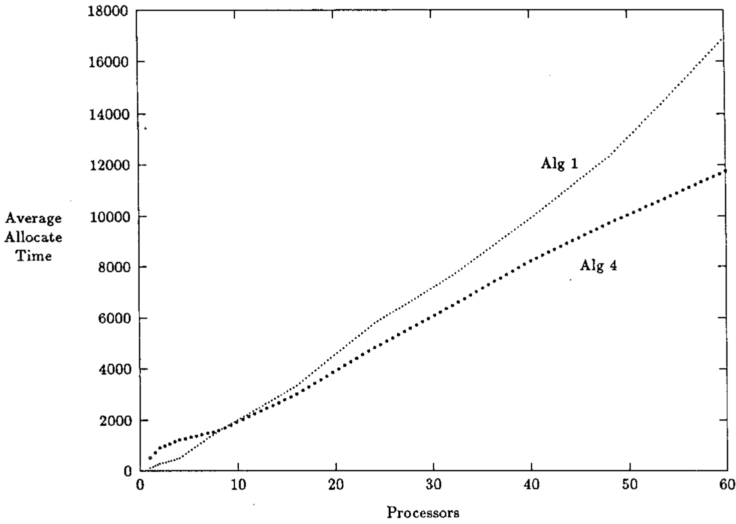

Fig. 8. Average time for allocation operation (in microsec.) in steady state.

During the startup phase, allocations compete only with other allocations starting with the initial state of the freelist which represents one large contiguous block. Figure 10 compares the cost to allocate for each of the three solutions implemented. The situation is quite different from the steady state case. Algorithm 1 has the lowest cost up to approximately

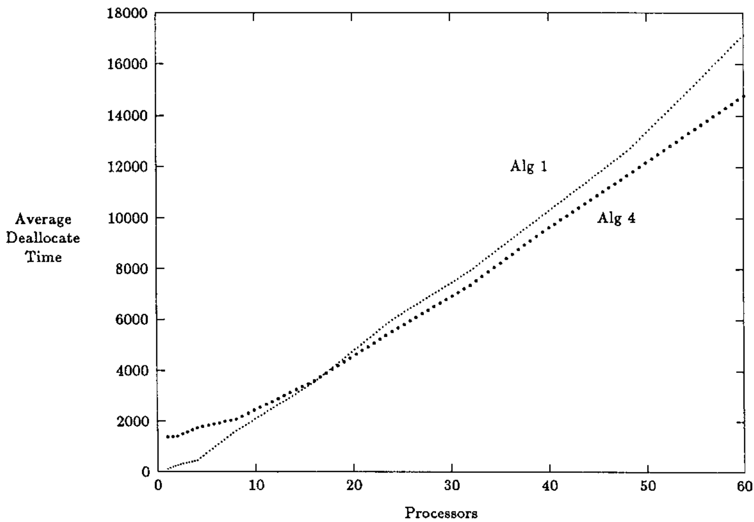

Fig. 9. Average time for deallocation operation (in microsec.) in steady state.

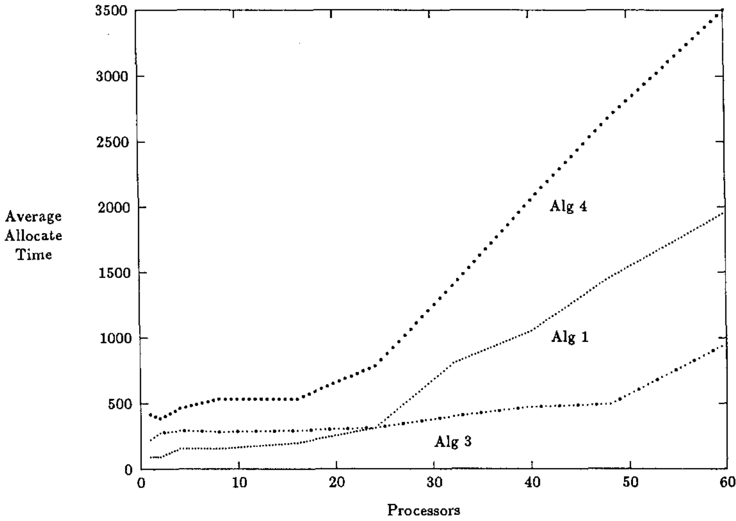

Fig. 10. Average time for allocation operation (in microsec.) during startup phase.

24 processors and then algorithm 3 becomes the best choice. Even with the lock on the entire sublist, the operation of algorithm 1 is fast enough (just subtracting the requested size from the size field of the one and only block) that waiting for access to the freelist is not significant. The locking overhead associated with algorithm 3 is minimized because the freelist is short

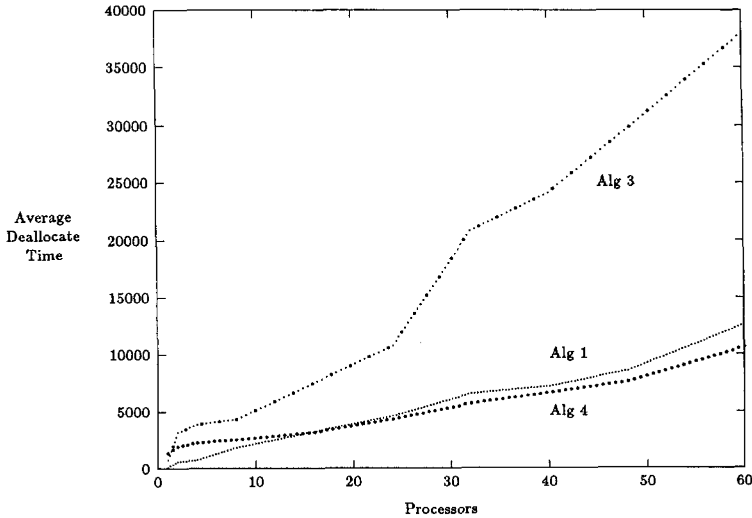

Fig. 11. Average time for deallocation operation (in microsec.) during windup phase.

(only one node to lock), the implementation of the read-lock operation is efficient (it is the repeated application of it that hurts in the steady state case), and there are no incompatible locks. The parallelism of the finer granularity access in algorithm 3 produces good results up to around 50 processors where contention appears to become a problem. Now it is algorithm 4 that is not competitive (the implementation of its synchronization is less efficient).

Figure 11 shows the average cost of deallocating blocks in the windup phase. Algorithm 3 suffers as it did in steady state. Algorithms 1 and 4 show the same kind of crossover as in steady state. However, if applications were somehow limited to the two-phase pattern, a hybrid approach might be worthwhile: using algorithm 3 during the allocation phase, with a clean transition to algorithm 1 for deallocation. Although algorithm 4 is still marginally superior to algorithm 1 in larger processor configurations, it does not use a compatible freelist structure for this kind of hybrid approach.

4. SUMMARY

In this paper, we have discussed four different approaches for dynamic memory allocation in a shared memory multiprocessing environment. There are really two important aspects to this work. One part is concerned with the investigation of design techniques that allow potentially much more parallel access to be accommodated within the components of a complex data structure. The second aspect deals with the mapping of these parallel data structures on real rather than ideal parallel architectures.

Much of the work with concurrent data structures investigates subtle techniques and design decisions that allow processes to coexist with shared data structure components and, often, in spite of changing values and dynamic structure. Some examples arise in our solutions. The **TryToAllocate** and **TryToAppend** routines use the atomic fetch-and-Φ on the size field of a freelist node for synchronization as well as encoding the available space information. The two roles are particularly compatible in this case. Processes can interpret the returned values to determine its current role. In contrast, we see that there is a price to pay in having header data within the allocatable free space itself. The two roles played by the same space in the case of algorithm 3 incur more locking overhead for coordination than in the other node granularity solution, algorithm 4. Algorithm 4 achieves the elimination of lock-coupling in the search phase by adopting a "sloppier" data structure with the presence of deleted header nodes and the possibility of a process encountering obsolete information. The last two solutions illustrate both the utility and the complexity of using

various kinds of compatible locks in trying to increase the level of potential parallelism. As parallel machines become more prevalent, the value of those techniques in the design of highly parallel algorithms could become extremely significant.

The motivation for the experimental work has been to evaluate these solutions in a realistic architecture and determine for what kinds of workloads (i.e. mix of operations, number of processes) each of the algorithms can be effective. The results are not surprising. As expected, the simple approach (algorithm 2) is best at light loads. Therefore, this is the appropriate solution for the level of parallelism supported by environments like the Uniform System where the number of memories and the number of processes are approximately equal. As the number of processors requesting space from a memory grows, the finer granularity algorithms are beneficial, to a point. The 2-phase request pattern does suggest a significantly different choice from the pattern consisting of a balanced mix of operations.

It was a fortunate set of circumstances that led us to gather results from the two related but, for this application, significantly different architectures. Having to interpret and reconcile the differences in the results from the two machines has provided interesting insights into the balancing of the local/remote access time ratio and the proportion of local/remote access within an algorithm.

REFERENCES

1. D. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching (1973).
2. C. Ellis, Concurrent Search and Insertion in 2–3 Trees, *Acta Informatica* **14**:63–86 (1980).
3. C. Ellis, Concurrency in Linear Hashing, *ACM Trans. Database Systems* **12**(2):195–217 (1987).
4. C. Ellis, Concurrency in Extendible Hashing, *Information Systems* **13**(1):97–109 (1988).
5. C. Ellis, Concurrent Search and Insertion in AVL Trees, *IEEE Transactions on Computers* **C-29**(9):811–817 (September 1980).
6. P. L. Lehman and S. B. Yao, Efficient Locking for Concurrent Operations on B-Trees, *ACM Trans. Database Systems* **6**(4):650–670 (December 1981).
7. D. Shasha and N. Goodman, Concurrent Search Structure Algorithms, *ACM Trans. Database Systems* Vol. 13, No. 1 (March 1988).
8. H. Stone, Parallel Memory Allocation using the FETCH-AND-ADD Instruction, Technical Report RC 9674, IBM Thomas, J. Watson Research Center, (November 1982).
9. A. Gottlieb and J. Wilson, Using the Buddy System for Concurrent Memory Allocation, Ultracomputer System Software Note 6, Courant Institute, New York University, (April 1981).
10. A. Bigler, S. Allan and R. Oldehoeft, Parallel Dynamic Storage Allocation, *Proceedings of Int'l Conf. on Parallel Processing*, p. 272–275 (August 1985).
11. R. Ford, Concurrent Algorithms for a Real Time Memory Management System, *IEEE Software*, p. 10–23 (September 1988).

12. H. T. Kung and J. Robinson, On Optimistic Methods for Concurrency Control, *ACM Trans. Database Systems* **6**(2):213–226 (June 1981).
13. A. Gottlieb and C. Kruskal, Coordinating Parallel Processors: A Partial Unification, *Computer Architecture News* **9**(6):16–24 (October 1981).
14. Butterfly Parallel Processor Overview, Technical Report, BBN Laboratories, (June 1985).
15. G. Pfister, W. Brentley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton and J. Weiss, The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, *Proc. of Int'l Conf. on Parallel Processing*, St Charles Illinois, p. 764–771 (August 1985).
16. Inside the Butterfly Plus, Technical Report, BBN Laboratories, (October 1987).
17. The Uniform System Approach to Programming the Butterfly Parallel Processor, Technical Report # 6149, BBN Laboratories, (October 1985).
18. H. T. Kung and P. L. Lehman, Concurrent Manipulation of Binary Search Trees, *ACM Trans. Database Systems* **5**(3):339–353 (1980).
19. U. Manber and R. E. Ladner, Concurrency Control in a Dynamic Search Structure, *ACM Trans. Database Systems* **9**(3):439–455 (September 1984).