A Concurrent Fast-Fits Memory Manager
University of Florida, Dept. of CIS Electronic TR91-009

Theodore Johnson
Dept. of Computer and Information Science, University of Florida
ted@cis.ufl.edu

September 12, 1991

Abstract

Shared memory multiprocessor systems need efficient dynamic storage allocators, both for system purposes and to support parallel programs. Most memory manager algorithms are based either on a free list, which provides efficient memory use, or on a buddy system, which provides fast allocation and release. In this paper, we present two versions of a memory manager based on the fast fits algorithm, which keeps free memory blocks in a Cartesian tree. A fast fits memory manager provides both efficient memory usage, and fast allocate and release operations. The concurrent implementations of the fast fits algorithm range from a simple moderate concurrency solution to a more complex but high concurrency solution.

1 Introduction

A *memory manager* accepts two kinds of operations: requests to allocate and requests to release blocks of memory, which may be of an arbitrary size. For example, the UNIX system calls `malloc()` and `free()` are requests to a memory manager. A concurrent memory manager handles requests for shared memory in a multiprogrammed uniprocessor, a shared memory multiprocessor, or a distributed shared virtual memory environment. In order to be correct, a concurrent memory manager should correctly allocate and release memory, merge adjacent free blocks, and allocate a block of memory in response to an allocate request if a sufficiently large block is available and isn't claimed by another allocate request. A simple concurrent memory manager might consist of a serial memory manager placed in a critical section. While this algorithm would be sufficient for a multiprogrammed uniprocessor, it can create a serial bottleneck in a parallel processor. An example application of parallel memory managers is parallel sparse matrix factorization algorithms [4, 5].

Most heap memory management algorithms use one of two main methods: free lists and buddy systems. In a *free list* algorithm [12], the free blocks are linked together in a list, ordered by starting address. Initially, all of the memory is free, and the free list consists of a block containing the entire memory. An allocate process searches the list until it finds the most promising block. The allocate process then takes the requested memory from that block, and removes the block from the list if the block becomes empty. A release process searches the list for the position to return the block. If possible, the release process merges the block being freed with blocks already on the list. Otherwise, the block is inserted into the free list. The method by which the allocate process finds the most promising block determines the free list algorithm. Concurrent free list managers typically use *first fit* : that is, they allocate from the first block that is large enough. This algorithm is used because only local information is needed to decide whether or not to allocate from a block.

One problem with a free list memory manager is that a process must search the free list in order to release or allocate a block of memory, and the free list might be quite long. An alternative is to use a

buddy system [13]. In a buddy system, memory blocks are available only as one of several fixed sizes. Each memory block has a buddy, with which it can combine and form a larger size block. The available sizes are determined by the choice of buddies. In a binary buddy system, for example, a block and its buddy are the same size. When the buddies combine, they form a buddy of the next larger size, so that the size of binary buddy blocks is $c2^i$, where size of the smallest allocatable block is c. Free blocks of size $c2^i$ are kept in the i^{th} free list. If no size $c2^i$ block is available to satisfy an allocate request, the system allocates a size $c2^{i+1}$ block and splits it, putting one half on the i^{th} free list, and using the other half to satisfy the memory request. Although buddy systems execute quickly, they are tend to waste space due to internal fragmentation. Fibonacci buddy systems were introduced to offer more block sizes and reduce internal fragmentation, but they suffer from external fragmentation due to unused block sizes.

In [16, 17], Stephenson proposes a fast free list memory management algorithm which he calls *fast fits*. In the fast fits algorithm, the memory blocks are linked into a Cartesian tree, with one coordinate being the physical location and the other being the size of the block. Since the blocks are organized into a binary tree, a block can be allocated or released in an expected $O(\log n)$ time. The blocks in the tree may be of any size (up to the size of the shared memory), so fast fits algorithm doesn't suffer from internal fragmentation. The fast fits algorithm is also well suited to supporting concurrent operations, because the tree structure will allow memory allocate and release requests to traverse separate branches, so that many operations can work on the tree concurrently. The tree structure of fast fits also allows the use of concurrency control algorithms developed for tree data structures [2, 6]. The fast fits memory manager is used in a number of commercial operating systems, including SUN Microsystem Unix[1].

A number of concurrent free list algorithms have been proposed. Stone [18] proposes a first-fit free list algorithm that uses the fetch-and-add instruction [9] and locking for concurrency control. Memory is added to or removed from a free block using the fetch-and-add instruction, but blocks must be exclusively locked whenever a free block is added to or removed from the list. Bigler, Allan and Oldehoeft [3] compare three algorithms, one of which is a concurrent algorithm that searches the free list using lock-coupling (the successor block must be locked before the lock on the current block may be released). The authors find that the concurrent algorithm is more appropriate for a parallel processing environment than are the serial algorithms. Ellis and Olson [7] propose two concurrent free list algorithms. Their first algorithm is similar to Stone's [18], but breaks the memory being managed into several segments, each of which has an independent free list. Ellis and Olson's second algorithm greatly simplifies the locking performed on the free blocks, and keeps around empty blocks until it is guaranteed that no operation will read the header information for that block. Ford [8] discusses several real-time memory managers, comparing a locking approach to an optimistic approach. In Ford's algorithms, memory is kept in a free list, with an index to the most recently released blocks. The algorithms allow the release operations to finish quickly at the expense of the allocate operations. Ford found that the optimistic memory manager is superior to the locking approach because the critical sections could be much shorter, allowing high-priority operations to gain control of the critical sections sooner.

Gottlieb and Wilson developed concurrent buddy systems that use fetch-and-add to coordinate processors. Their first algorithm [10, 20] considers a buddy system to be organized as a tree. A count of the number of blocks of each size that are contained in subtree rooted at a node is stored at each node. Concurrent allocators use this information to navigate the tree. Their second algorithm [11, 20] is a concurrent version of the commonly described buddy algorithm. This algorithm uses semaphores to enforce concurrency control.

All previously published concurrent memory manager algorithms are based on either a free list or a buddy system. In this paper, we develop two concurrent fast fits based memory managers, in order to take advantage of the fast fits algorithm's properties of fast allocation and release, and efficient memory use.

[1] as noted in the `malloc` manual page

2 The Fast Fits Memory Manager

The fast fits memory manager keeps the free blocks in a Cartesian tree [19, 1]. A Cartesian tree is a binary tree that stores points from the two-dimensional plane. Each node in the tree consists of a single data item and pointers to two children. Each data item consists of an X and a Y coordinate and possibly some associated information. If n is a node in a Cartesian tree, let $n.X$ be the stored X coordinate and let $n.Y$ be the stored Y coordinate. The nodes in the Cartesian tree are ordered by the following two rules (see figure 1):

1. If $n_1.X < n_2.X$, then n_1 comes before n_2 in the inorder listing of the Cartesian tree.
2. If n_1 is a descendant of n_2, then $n_1.Y \leq n_2.Y$.

One can implement search, insert and delete operations on a Cartesian tree using local rebalancing only [19, 18]. For a given set of points with unique X and unique Y values, there is a unique Cartesian tree that contains those points, so the rebalancing involves restoring the structural properties of the tree. Vuillemin [19] shows that the expected number of comparisons needed to insert a point into a Cartesian tree is $O(\log n)$ (assuming that the Y components of the entries in the tree form a random permutation).

In the fast fits algorithm, the X coordinate corresponds to the starting address of the free block and the Y coordinate corresponds to the size of the free block. The information needed to maintain a node that corresponds to a free block is kept at the beginning of a free block, thus placing a limit on the minimum size block that may be allocated. The fast fits data structure needs to modify the Cartesian tree insert and delete operations to produce release and allocate operations. We define the *neighbors* of a block of memory to be the blocks of memory that are adjacent to it. A release operation must search for its neighbors before inserting the released block of memory. The allocate operation must follow an allocation strategy to find a suitable block of memory, then either allocate the entire free block (and delete the block), or allocate part of the free block (and demote the node) [18].

3 The Algorithms

In this section, we describe the concurrent fast fits memory manager algorithms. The algorithms increase in complexity, but also in the amount of concurrency allowed. The simpler algorithm also serves as a base for understanding the more complex algorithm.

3.1 W-only Algorithm

The first algorithm that we present places exclusive (W) locks, and uses lock-coupling (the child node must be locked before the parent can be unlocked) to prevent interference between operations [2]. The pseudocode for algorithm 1 is in the appendix.

We modify the data structure by adding an additional node to the data structure, the anchor node (see figure 2). The anchor node allows the pointer to the root to be locked, and simplifies restructuring operations involving the root. In addition, we modify the nodes in the tree by requiring that they contain the size of the child nodes as well as their locations. This information will simplify the task of finding a suitable child in the allocate procedure.

The allocate operation must find a suitable free block from which to allocate. Stevenson [18] recommends the *better-fit* algorithm. To allocate, the better-fit algorithm starts at the root of the Cartesian tree and examines the children of the root. If both children are smaller than the block to be allocated, the algorithm allocates from the root (if the root is too small, then the allocate operation fails). Otherwise, it goes to the smallest child that is at least as large as the block to be allocated, and repeats. Stephenson reports that the better-fit algorithm requires far fewer comparisons than the linked-list first fit algorithm. In order to gain concurrency, the operations should be hashed out among the subtrees as evenly as possible. For this reason, we use a slightly different algorithm, which we call *random better-fit*.

The difference between better-fit and random better-fit occurs when both children are large enough to allocate from. In this situation, better-fit will choose the smaller child, while random better-fit will choose either with probability 1/2.

The allocate operation begins by locking the anchor and the root of the Cartesian tree (which is the only child of the anchor). The procedure then applies the random better-fit search algorithm until a suitable node is found. If the current node has a child that is at least as large as the requested block, the operation releases the lock on the parent and locks the child selected by random better fit. Note that the children of the node under consideration don't need to be locked, since their size is stored in their parent's node. After the appropriate node is found, the operation has a W lock on the node and on the node's parent, and the requested block is allocated from the current node.[2] If the entire node is allocated, then the operation deletes the node from the tree. Otherwise the node needs to be demoted (pushed lower in the tree) if it becomes smaller than one or both of its children.

The delete procedure performs additional tasks related to the release operation, so we discuss it later. The demote procedure [18] restores the second property of the Cartesian tree (a block must be larger than its children). Due to the address ordering of the Cartesian tree, the demoted node will become the child of a node on the rightmost path of the left subtree, or on the leftmost path of the right subtree (see figure 3). The demote procedure combines these two paths, ordering by the size of the blocks. At every restructuring step, the demote procedure examines the `size` (or Y) component of the demoted node, the root of the right tree, and the root of the left tree; whichever is largest become the child of the parent. The initial right and left trees are the original right and left subtrees of the demoted node. Figures 3 and 4 illustrate the demote procedure. Thirty words have been allocated from node $(10, 54)$ of the tree in figure 1. In figure 3, the root of the right tree is attached to the parent because it is the largest block. In figure 4, the left tree is next added, then the demoted node. When the demoted node is added to the tree, the remaining right and left subtree is attached to the demoted node.

The demote procedure ensures that the node that it needs to modify is locked. Initially, this node is the parent; afterwards, it is the node most recently re-incorporated into the tree. The roots of the unincorporated right and left subtrees don't need to be locked because any operation that wished to modify their contents would need a lock on their parents, but these locks are held by the demote procedure.

The release operation returns a block of memory to the memory manager. In order to prevent fragmentation, the release operation must search for the neighbors of the blocked being released. There are two possible neighbors, and if they exist in the tree, they can be found by searching the tree for the address of the block being released until the neighbors are found or a leaf is reached [18], so the release operation can search for the neighbors while it searches for the position to insert the released block. The largest neighbor might be larger or smaller than the block being released; correspondingly the release operation will take one of two actions.

The release operation starts by locking the anchor and making the root the current node. At each search step, the release operation examines the size and position of the current node (which is stored in the locked parent). If the current node is a neighbor of the released block, or if the current node is smaller than the released block, the release operation terminates its search and locks the current node, otherwise it makes the node the parent and picks the next node based on address ordering. After locking the node, the operation checks if it's a neighbor of the released block. If so, the operation deletes the current node, combines the two blocks, then releases the combined block. The delete procedure also searches the subtree for neighbors.[3] Otherwise, the operation inserts the released block between the parent and the node. The insert procedure also searches for neighbors, and might find two.

The delete procedure is similar to the demote procedure, as again the rightmost path of the left

[2] Allocating from the end of the block is easier to implement because the position of the Cartesian tree node information doesn't move. Allocating randomly from either end will produce a better branching tree, but the underlying system might constrain the implementation to allocating from a particular end.

[3] An optimization, discussed later, is to merge the two blocks if their combined size is less than the parent's size, then search the subtree for a neighboring block.

subtree and the leftmost path of the right subtree are merged, the difference being that only two nodes are compared (as illustrated in figures 3 and 4) The neighboring nodes of the child node (which was combined with the released node) are detected and combined with the child. If neighbors of the deleted node are in the subtree, they will either be the last node on the rightmost path of the left subtree, or the last node on the leftmost path of the right subtree (see figure 3). These nodes are easy to delete, because they will have only one child, which can be attached to the parent in place of the node.

The insert procedure is slightly different in that it must create the leftmost path of the right subtree and the rightmost path of the left subtree, rather than merge it. The parent of the node being inserted is assumed to be locked when the procedure is entered. The inserted node is made a child of the parent, displacing the subtree that is to be split. The inserted node is the last node on the initial leftmost path of the right subtree and also the last node on the initial rightmost path of the left subtree, called `righthook` and `lefthook` respectively. On each step, the root of the displaced subtree is attached to `righthook` or `lefthook` depending on whether its X value is larger or smaller than that of the inserted node. The root of the displaced subtree then becomes `righthook` (`lefthook`), and the left (right) subtree becomes the displaced subtree. The steps of an insert procedure are shown in figures 5 and 6. The node $(9.5, 30)$ is inserted into the tree left after demoting node $(10, 20)$. In figure 5, the inserted node initially contains the `lefthook` and `righthook` pointers. Since the root of the displaced subtree is to the left of the inserted node, it is attached to `lefthook`, and becomes the rightmost node of the left subtree. In figure 6, the child is to the right of the inserted node, so it is attached to `righthook`. After this step, the detached subtree is empty, so the restructuring is finished.

Both `righthook` and `lefthook` are always W-locked, and the procedure applies lock-coupling by locking the new `righthook` (`lefthook`) before releasing the lock on the old one. If the insert procedure detects a neighbor of the node being inserted, then that node will be the last node on the right(left)most path of the left(right)subtree. The left subtree of the neighbor will be to left of the node being inserted, and the right subtree to the right, so the children can be immediately attached. All that remains is to search for the other neighbor.

3.1.1 Correctness

We show that a fast fits operation always sees a correctly structured tree in the sense that the nodes that it reads satisfy the two ordering conditions listed in section 2. If the operations are executed serially, then the operations see a correctly structured tree because the operations restructure the tree so as to to satisfy the node ordering conditions.

Let us order the operations on the tree by the time at which they lock the anchor, so we'll say that $O_1 < O_2$ if O_1 locked the anchor before O_2 did. Let us define the *locked path* of operation O to be the set of nodes that it locks, and let us define the *intersecting path* of O_1 and O_2 to be the set of nodes that both O_1 and O_2 lock. Since the operations place exclusive locks using the lock-coupling protocol, if $O_1 < O_2$, then O_1 locks the nodes on the intersecting path of O_1 and O_2 before O_2 does. Finally, let us define the *keyrange* [15] of a node to be the set of keys ((X, Y) pairs) that can exist in the subtree rooted at the node.

Let the keyrange of a node, n, on operation O_1's locked path be the bounds on the set of keys that might exist at the subtree rooted at n, $X_{lo} < X < X_{hi}$, and $Y \leq Y_{hi}$. As O_1 traverses the tree from the root to the leaf, we can imagine that it calculates the keyrange of the current node. We'll say that O_1 is *correctly navigated* if the calculated keyrange of a node, n is the same as the actual keyrange of n. Since operations are ordered as they traverse the tree, if $O_1 < O_2$, then O_2 won't overtake O_1 as they traverse the tree. Therefore, if O_1 always reads a correctly structured subtree, then O_1 is correctly navigated.

Since the operations are correct if they execute serially, the modifications that operation O makes to its locked path are correct. Let l and r be the left and right children of n, where n is a node on operation O_1's locked path. After operation O_1 unlocks n, it will never again modify n. Therefore, after O_1 unlocks n, the keyrange of n, l, and r are correct because O_1 correctly navigated and correctly restructured the tree. Therefore, if O_1 correctly navigates, and O_2 is the first operation to lock node n after operation

	R	W	U
R	yes	no	no
W	no	no	no
U	no	no	no

Table 1: Lock Compatibility Chart

O_1 unlocks it, then O_2 reads a correctly structured subtree.

This argument leads to:

Theorem 1 *Every operation correctly navigates and sees a correctly structured tree.*

Proof: We prove the claim by induction. For the base case, consider the first operation on the fast fits data structure. The data structure consists of a single node, the root, that represents all of allocatable memory. This tree is correctly structured, so the first operation correctly navigates and leaves a correctly structured subtree.

For the inductive case, assume that all operations up to the i^{th} left a correctly structured subtree at all nodes on their locked paths. When the i^{th} operation reads the first node on its locking path, it reads a node that was last locked by the j^{th} operation, $j < i$, Therefore, operation i reads a correctly structured tree, so it will correctly navigate to the next node and will leave a correctly structured subtree when it unlocks the node. By induction, operation i correctly navigates to every node in its locking path, As a result, operation i leaves a correctly structured subtree at every node that it unlocks, proving the induction •

To finish the showing that the W-only algorithm is correct, note that the algorithm is deadlock-free because locks are always placed top-down. Since the tree is correctly structured, a release operation always finds the neighbors that are placed in the tree by a preceding operation, so that all neighbors are eventually merged. Since the largest free block is always at the root, an allocate operation returns successfully if a sufficiently large block of memory exists in the data structure.

3.2 The RWU Algorithm

The W-only algorithm permits several operations to execute concurrently because the operations may travel to different branches of the Cartesian tree. However, the root is a serialization bottleneck since every operation must place an exclusive lock on the root at some point. By requiring the exclusive lock on the root, the W-only algorithm orders every operation on the tree. This is a stricter ordering than is necessary, since two operations might restructure different subtrees, and the ordering is only necessary among operations with conflicting read and write sets. The second algorithm that we propose makes an initial *optimistic descent* [2] in which it places shared (R) locks until it finds a node that will be involved in restructuring, at which point it places a write-upgrade lock (U lock), and exclusive locks after that. A lock compatibility chart is listed in table 1. The R and W locks are the usual read and write locks, and are granted in FCFS order. An R lock can be upgraded to a U lock, which will be granted before any W locks are granted, but after all previous U locks and currently held R locks are released. We require the U locks to have a higher priority than the W locks in order to enforce an ordering between operations that have conflicting read and write sets.

The code for the allocate and release operations is in the appendix. The release algorithm starts by placing an R lock on the anchor. The operation then searches for the root of the subtree that it needs to restructure (due to finding either a neighbor or a smaller free block on its path), making use of the information about the child that is stored in the parent. After the root of the subtree is found, it must be U locked. When the release operation places its U lock on node n, it establishes its position in the ordering of all operations whose locked path includes n. When an operation attempts to place a U lock,

other operations might have U locks pending. As a result, when the operation obtains its U lock on the node, the children of the node might have been modified (although the locked node won't be modified because U locks have priority over W locks). Therefore, the operation must check the node and the child to determine whether an operation can still be performed (i.e, check if the targeted child is neighbor, or is smaller than the block being released). If neither a delete nor an insert can be performed, the release operation must continue to search downwards for the place to release the block, using W locks.

The allocate operation searches for the node from which to allocate in the same manner as in the W-only algorithm. Since the operation needs to know the sizes of the children of a node to determine whether to allocate from the node, the node must be locked when it is examined. When a suitable node is found, the parent must be U-locked. The operation must first release the lock on the child in order to prevent deadlock. Again, another operation might obtain its U lock before the allocate operation obtains its U lock, so the allocate operation must check whether one of the parent's children is large enough to allocate from. If not, the allocate operation must release its locks and start over using the W-only algorithm.[4] Otherwise, the operation W-locks a child that is large enough and continues searching, since the child might also have children that are large enough.

3.2.1 Correctness

The W-only algorithm orders all operations by the time at which they lock the anchor. The RWU algorithm gains concurrency by loosening the operation ordering, and as a result one operation might overtake another while they read the data structure. In order to analyze the RWU algorithm, we need to establish an ordering between operations that have read-write conflicts on a node. We will say that O_1 *precedes* O_2 at node n if O_1 places lock L_1 on n before O_2 places lock L_2 on n, and 1) L_1 is a U or W lock, and O_2 never places a U or W lock on n, 2) O_1 never places a W or U lock on n, and L_2 is a U or W lock, or 3) both L_1 and L_2 are U or W locks. Since the operations use lock-coupling, if O_1 precedes O_2 at n, then O_1 precedes O_2 at every node in their intersecting path in which at least one of the operations places a U or W lock. Therefore, we will say that $O_1 <_p O_2$ if O_1 precedes O_2 at some node in their intersecting path.

In order to apply the arguments used to show the correctness of the W-only algorithm, we need a total order on the operations, but we can use any total order $<$ that is consistent with the partial order $<_p$. We can now apply the arguments from the W-only algorithm to see that every operation that sees a correctly structured tree correctly navigates, and, because of the consistency check after obtaining the U lock, every operation that correctly navigates to a node and places a W or U lock on it correctly restructures the subtree rooted at that node. We can then conclude that every operation correctly navigates and correctly restructures the tree.

To finish the correctness argument, we note that the algorithm is deadlock-free because the only time when an operation places a lock on the ancestor of a node that it has locked is when the allocate operation U-locks the parent of the node that it plans to allocate from. At this point, the allocate operation holds only the lock that it is upgrading, breaking the circular wait. While operations can be forced to restart, the algorithm is made starvation-free by allowing operations to use the W-only algorithm after a finite number of restarts.

3.2.2 Implementing U Locks

Assuming that U locks are available to the implementer simplifies the description of the algorithm. However, their semantics are unusual, so that while R and W locks should be available, it is unlikely that U locks will be available. We next describe a simple method to implement the equivalent of U locks by using R and W locks and examining the lock queue.

[4] An optimization, examined later, is to allow k restarts using the RWU algorithm before resorting to the W-only algorithm, trading variance in the execution time for increased concurrency.

For this implementation, we assume a spin-lock implementation of the R and W locks in which the head of the queue can be read by the processes, such as the one described by Mellor-Crummy and Scott [14]. The key to the simulation is the observation that in the RWU algorithm, at most one W lock will be in a node's lock queue at a time (except for the anchor, where there is no problem).

The processes use the following protocol to place locks. In order to upgrade from a R lock to a U lock, the process enqueues a W lock, then releases its R lock. In order to set a W lock, the process simply enqueues the W lock. After obtaining the W lock, the process checks to see if its lock is the only one in the queue. If so, the process proceeds. Otherwise, the process releases the lock and places it again.

Since an operation that places a W lock must have a lock on the parent of the node, no more operations will join the node's lock queue, so the W-locking operation won't starve. If a process obtains a W lock on a node, and there are other locks in the node's queue, the other lock requests must be U locks. By releasing and relocking the node, the W-locking operation flushes out the U locks, and gives them priority.

4 Performance

We wrote a concurrent fast fits simulator to study the performance of the algorithms and their optimizations. The simulator starts with all of memory free. Allocate requests arrive according to a Poisson process. The allocated block is released after an exponentially distributed length of time. Each node access time is exponentially distributed.

Our first set of experiments compared the performance of the different algorithms. Figures 7 and 8 show the response times of the allocate and the release operations, respectively, with an increasing arrival rate. Each node access requires an expected one time unit. The total memory size is 2M (2^{21}) words, and the average request size is 500 words. Half of the request sizes were chosen from a uniform distribution, and half from an exponential distribution (truncated to the memory size). A block is released an expected 5000 time units after being allocated. The simulator allocated and deallocated 100,000 blocks before terminating, the statistics were collected for the last 80,000 operations only.

The first algorithm that we simulated is the W-only algorithm. The response time of the allocate operations is directly measurable, but a release operation might make several passes. The curve for the W-only algorithm in figure 8 is the response time for a single pass of a release operation multiplied by the average number of passes per block release. In this experiment, each block release required about 1.94 passes.

The large number of release operations per block deallocation makes the W-only algorithm inefficient, and reduces the maximum arrival rate. In order to reduce the number of passes, we implemented in the release operation the optimization (noted in footnote 3) where instead of always deleting a neighbor found during the release operation, the merged block is only deleted if it is larger than the parent, otherwise it is left in the tree. If the merged block is left in the tree, the subtree must be searched for the other neighbor of the released block. If the neighbor exists, it will be the rightmost block in the left subtree or the leftmost block in the right subtree, depending on whether the released block was the left or right neighbor of the block in the tree. The appendix lists the code of the find operation, which looks for the neighbor and removes it from the tree if it exists. If the find operation removes a neighbor from the tree, the release operation must make another pass.

We ran simulations of the W-only algorithm with the find operation (W-only with Find), and plotted the results. The comparison of the allocate and release response times shows that using the Find operation significantly decreases the response times and increases the maximum throughput. The W-only with Find algorithm required a maximum of 1.32 release operations per block release, which accounts for the increased maximum throughput and the for the bulk of the decrease in the response time of the release operation.

Since the Find operation clearly improves performance, we only implemented the RWU algorithm with finds, which is also plotted in figures 7 and 8. The response time of the RWU algorithm contains

very little waiting time, the increase in response time is due to an increase in the number of nodes examined per path. The number of allocated blocks ranged from a maximum of about 140 when the arrival rate is .02 to a maximum of about 1900 when the arrival rate is .34. The response curve ends well before the apparent onset of a serialization bottleneck. We could not get our simulator to successfully complete with a higher arrival rate, because of excessive lock queuing. When the simulator starts, the tree is small, and operations place W locks near the root. So, contention is high during the startup transient, although the contention disappears when the tree becomes large (the steady state).

All three concurrent algorithms compare favorably to executing the fast fits algorithm in a critical section. If the find operation is used, each allocate operation generates 1.3 release operations. The average path length for these operations is about 9, so it takes about 20 time units to allocate then release a block of memory. The knee in the response time curve would occur when the critical section has a 50% utilization, which would occur when the allocate operation arrival rate is about .025. The knee of the curve of the W-only with find algorithm occurs at an arrival rate five times greater.

We implemented the optimization of the RWU algorithm described in footnote 4. When an allocate operation places its exclusive locks and finds that it can't allocate a large enough block, it must release its locks and start the search over. The optimization we implemented (RWU-2 restart) executes a pass that uses the W-only algorithm the second time that it must restart. The 1-restart algorithm requires that .9% percent of allocate operations use the W-only algorithm when the arrival rate is .34, while the 2-restart requires that only .053% of the allocate operations use the W-only algorithm at the same arrival rate. The restart rate is very small for the 1-restart algorithm, and the 2-restart optimization had little effect on performance, so we don't report the results.

We ran another set of experiments that tested the memory efficiency of the fast fit algorithms. The arrival rate was held constant at .05, and the request size varied. The time between the allocation and the deallocation of a block was an expected 7500 time units. We increased the request size to simulate an increasing demand for memory, and plotted the abort rate due to lack of memory in figures 9 and 10.

In figure 9, we compare the memory efficiency of the W-only and the RWU algorithm using random better fit and also the RWU algorithm using better fit. Half of the block request sizes were uniformly distributed, half were exponentially distributed. Both random better fit algorithms had similar performance, requiring that almost 60% of memory be allocated before a significant number of allocate requests are denied. The better fit algorithm performed somewhat worse, denying significant numbers of requests when only 50% of memory was allocated. In figure 10 we compare the effects of the distribution of the block size requests. As can be expected, having all block size requests be exponentially distributed resulted in a higher abort rate than an mix of uniform and exponential, which was higher than the uniform-only abort rate.

5 Conclusion

We present two concurrent memory managers based on the fast fits algorithm. Both algorithms have good space efficiency, fast response times, and are easily implemented. The W-only algorithm supports moderate concurrency levels, and the RWU algorithm supports high concurrency levels. The parallelism available through the RWU algorithm increases as the fast fits tree increases. We examined several possible optimizations and found that the use of the *Find* operation yields a significant improvement, while allowing multiple restarts has little benefit. Random better fit allocation has space efficiency comparable to that of Stephenson's better fit, and even causes fewer aborts than better fit on the input distribution that we tested. The concurrent fast fits memory managers offer a good alternative to the existing concurrent buddy or concurrent free list algorithms.

6 Acknowledgements

We'd like to thank C.J. Stephenson for discussing his fast fits memory manager with us, and Tim Davis for his careful reading and advice.

References

[1] C. Aragon and R. Seidel. Randomized search trees. In *Proceedings of the 30th Symposium on the Foundations of Computer Science*, pages 540–545, 1989.

[2] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.

[3] B. Bigler, S. Allan, and R. Oldehoeft. Parallel dynamic storage allocation. In *International Conference on Parallel Processing*, pages 272–275. IEEE, 1985.

[4] T. A. Davis and P. C. Yew. A nondeterministic parallel algorithm for general unsymmetric sparse lu factorization. *SIAM J. Matrix Anal. Appl.*, 11(3):383–402, 1990.

[5] I. S. Duff. Multiprocessing a sparse matrix code on the Alliant FX/8. *J. Comp. Appl. Math.*, 27:229–239, 1989.

[6] C.S. Ellis. Concurrent search and insertion in AVL trees. *IEEE Transactions on Computers*, c-29(9):811–817, 1980.

[7] C.S. Ellis and T. Olson. Concurrent dynamic storage allocation - look it up. In *Proceedings of the international Conference on Parallel Processing*, pages 502–511, 1987.

[8] R. Ford. Concurrent algorithms for real time memory management. *IEEE Software*, pages 10–23, September 1988.

[9] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Coordinating large numbers of processors. In *Proceedings of the International Conference on Parallel Processing*. IEEE, 1981.

[10] A. Gottlieb and J. Wilson. Using the buddy system for concurrent memory allocation. Ultracomputer System Software Note 6, Courant Institute, 1981.

[11] A. Gottlieb and J. Wilson. Parallelizing the usual buddy algorithm. Ultracomputer System Software Note 37, Courant Institute, 1982.

[12] D. Knuth. *The Art of Computer Programming*, volume 1. Addison Wesley, 1968.

[13] J. l. Peterson and T. A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

[14] J.M. Mellor-Crummey and M.L. Scott. Synchronization without contention. In *Fourth Intn's Conference on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, 1991.

[15] D. Shasha and N. Goodman. Concurrent search structure algorithms. *ACM Transactions on Database Systems*, 13(1):53–90, 1988.

[16] C. J. Stephenson. Fast fits: New methods for dynamic storage allocation. In *Proc. of the Ninth ACM Symposium of OPerating System Principles*, pages 30–32, 1983.

[17] C. J. Stephenson. Fast fits: New methods for dynamic storage allocation. Technical report, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1983.

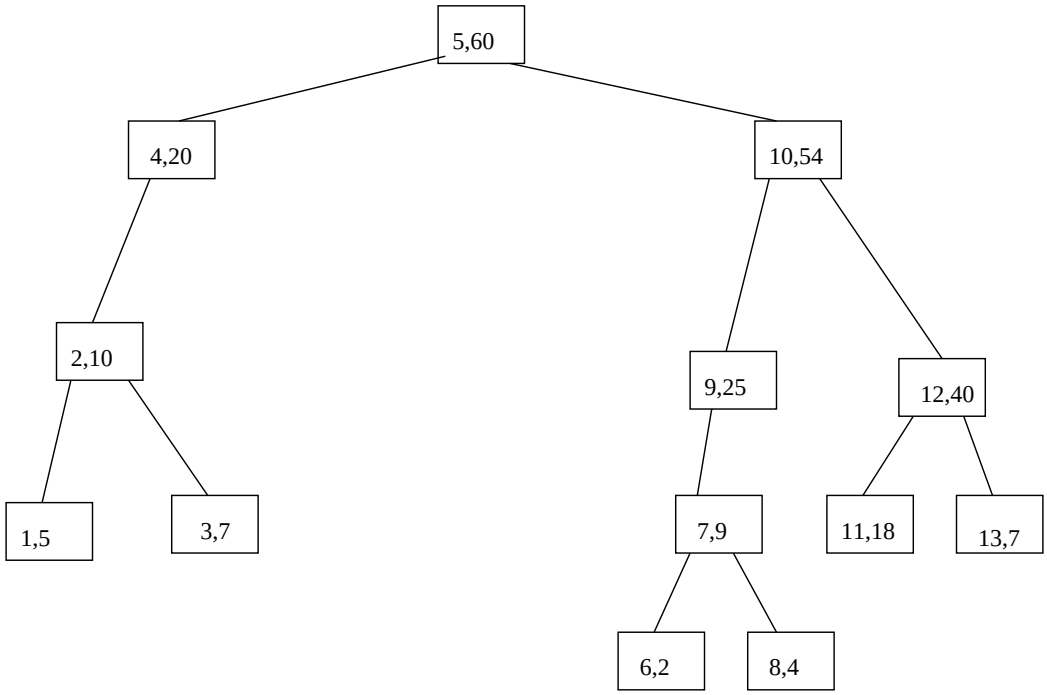

Figure 1: Example Cartesian tree

[18] H. Stone. Parallel memory allocation using the fetch-and-add instruction. Technical Report RC 9674, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1982.

[19] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

[20] J. Wilson. *Operating System Data Structures for Shared-memory MIMD Machines with Fetch-and-add.* PhD thesis, NYU, 1988.

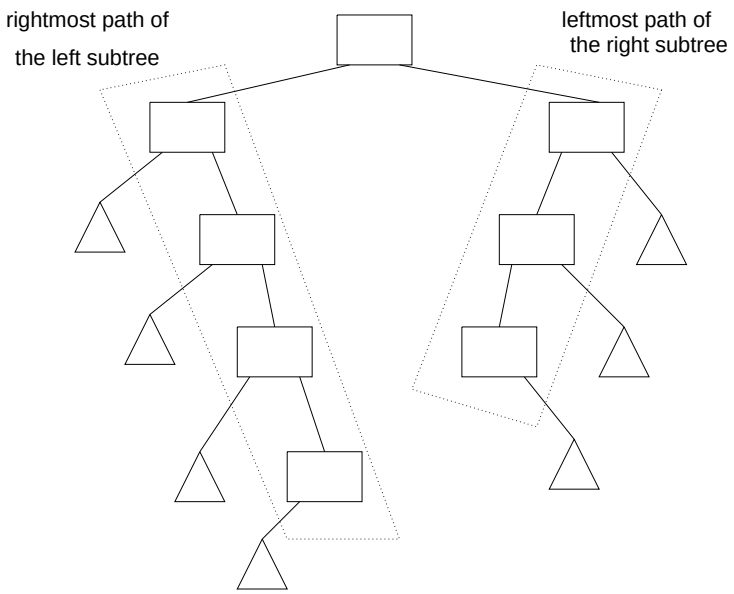

Figure 2: Right(Left)most paths in the left(right) subtree

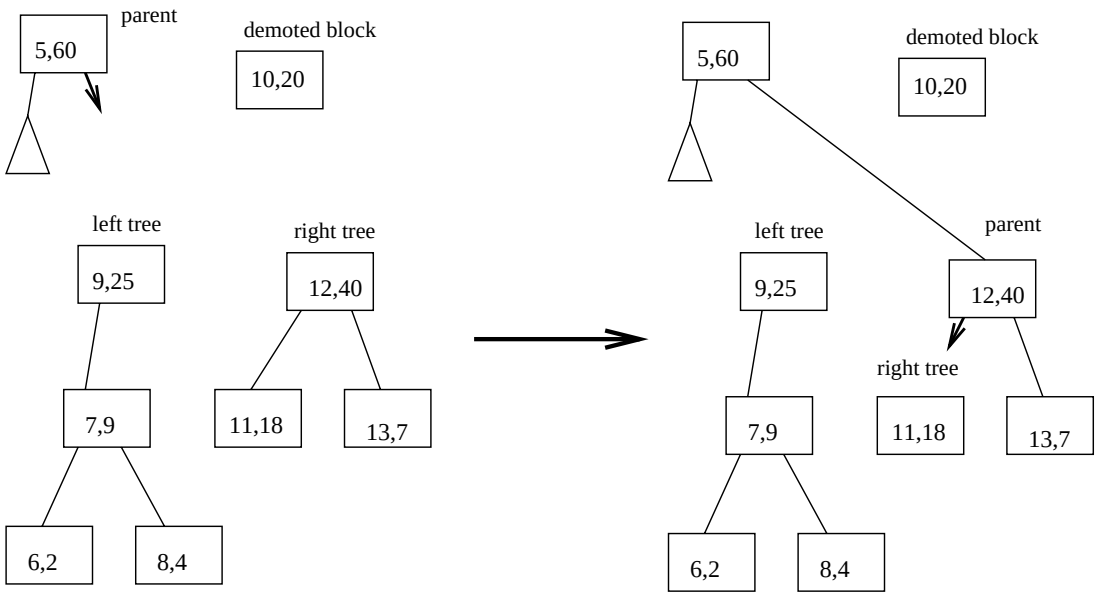

Figure 3: Demote operation restructuring example

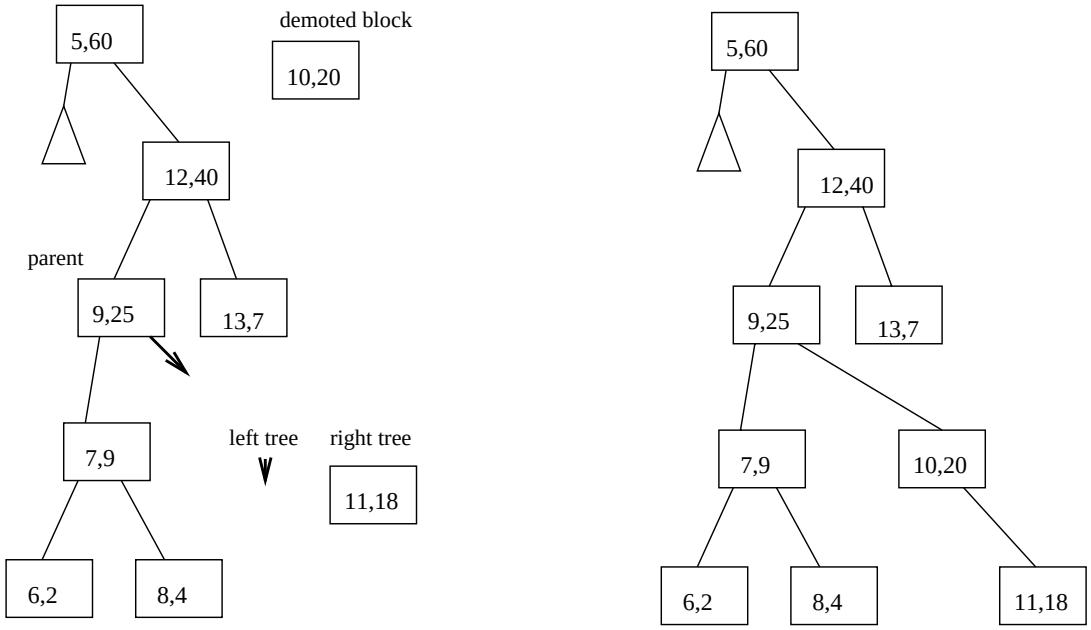

Figure 4: Demote operation restructuring example

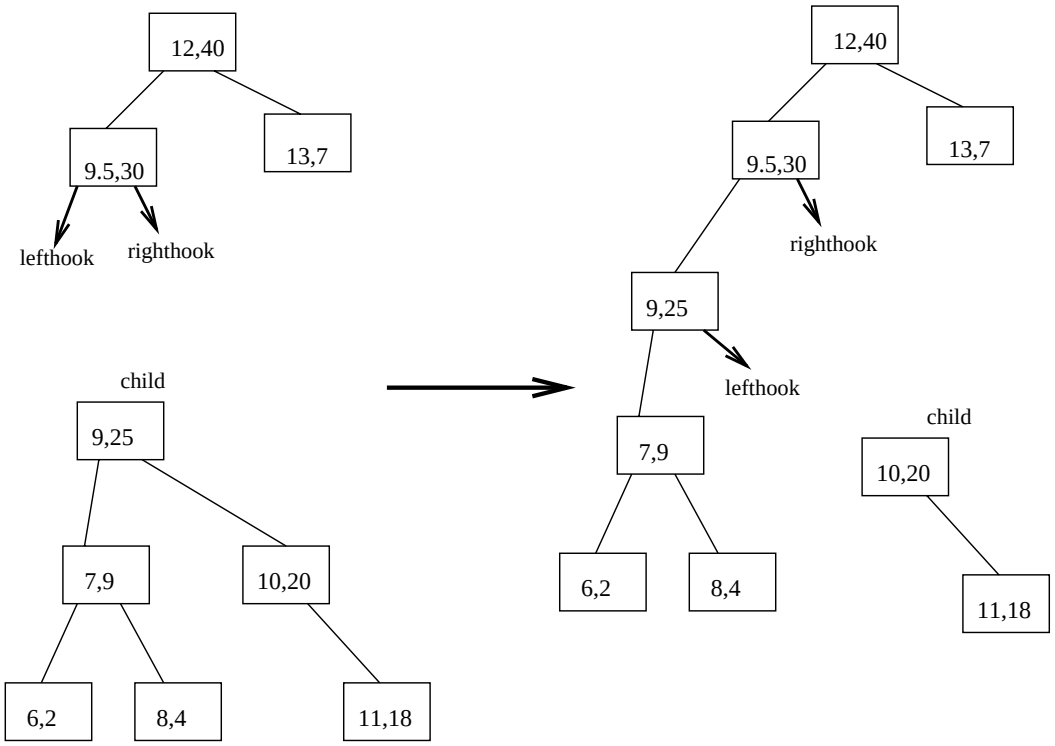

Figure 5: Insert operation restructuring example

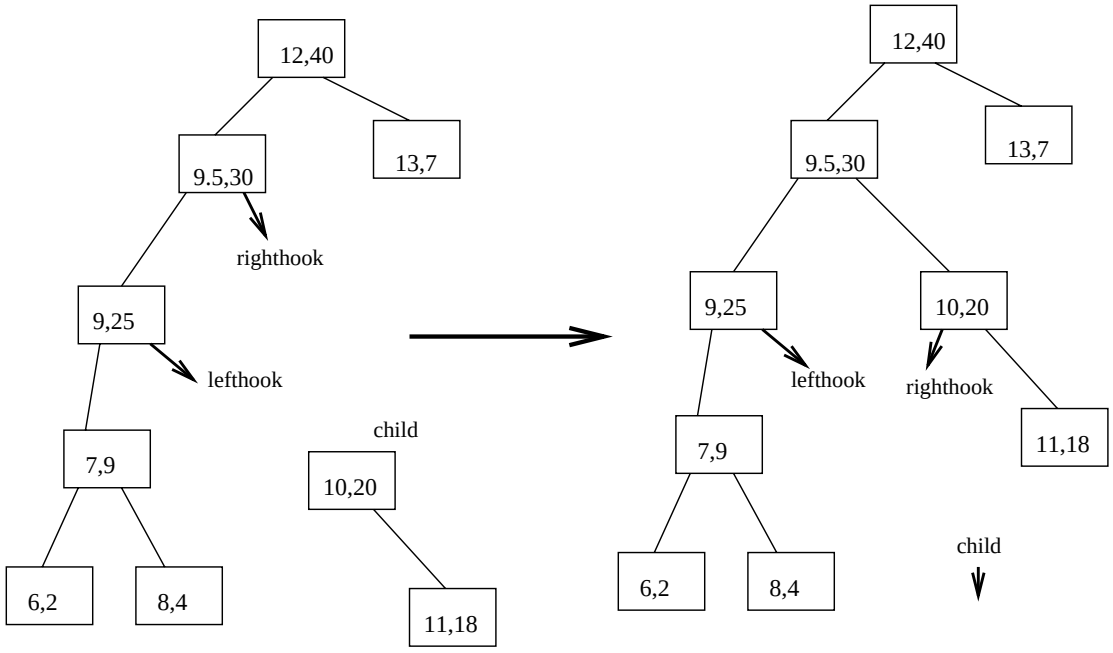

Figure 6: Insert operation restructuring example

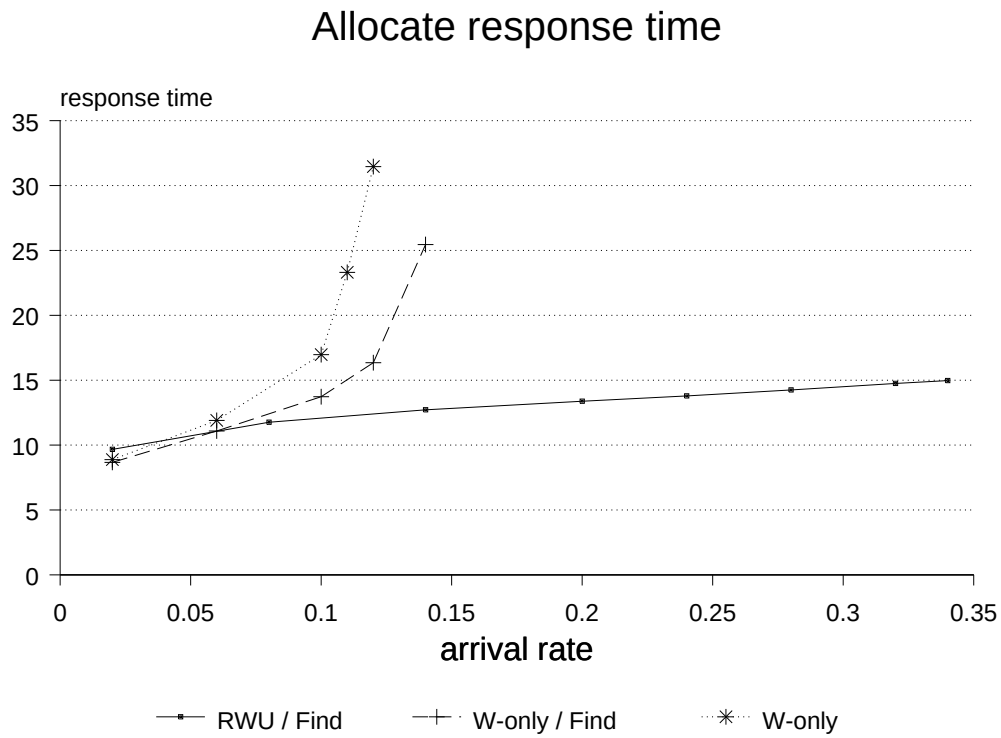

Figure 7: Response time vs. allocate operation arrival rate

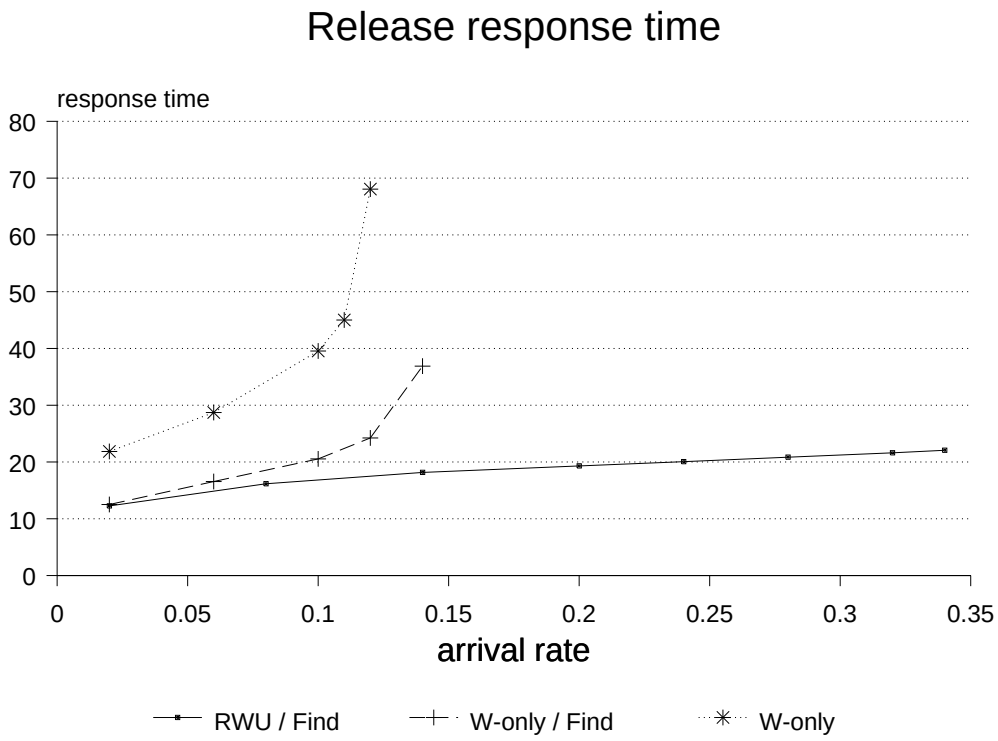

Figure 8: Response time vs. allocate operation arrival rate

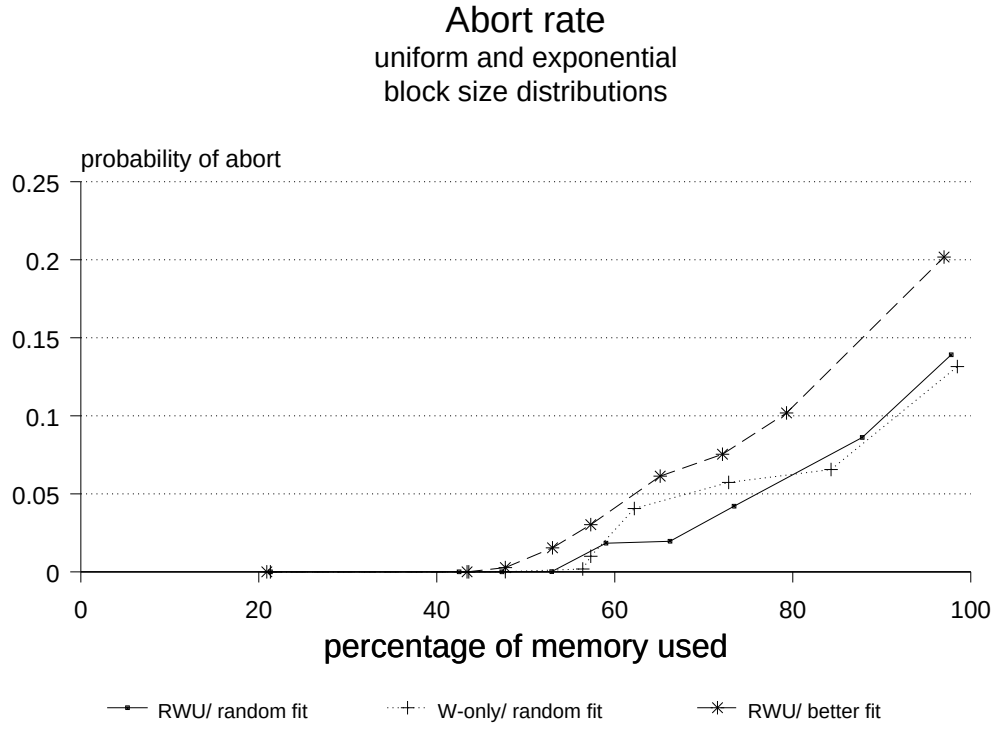

Figure 9: Comparison of space efficiency for different algorithms

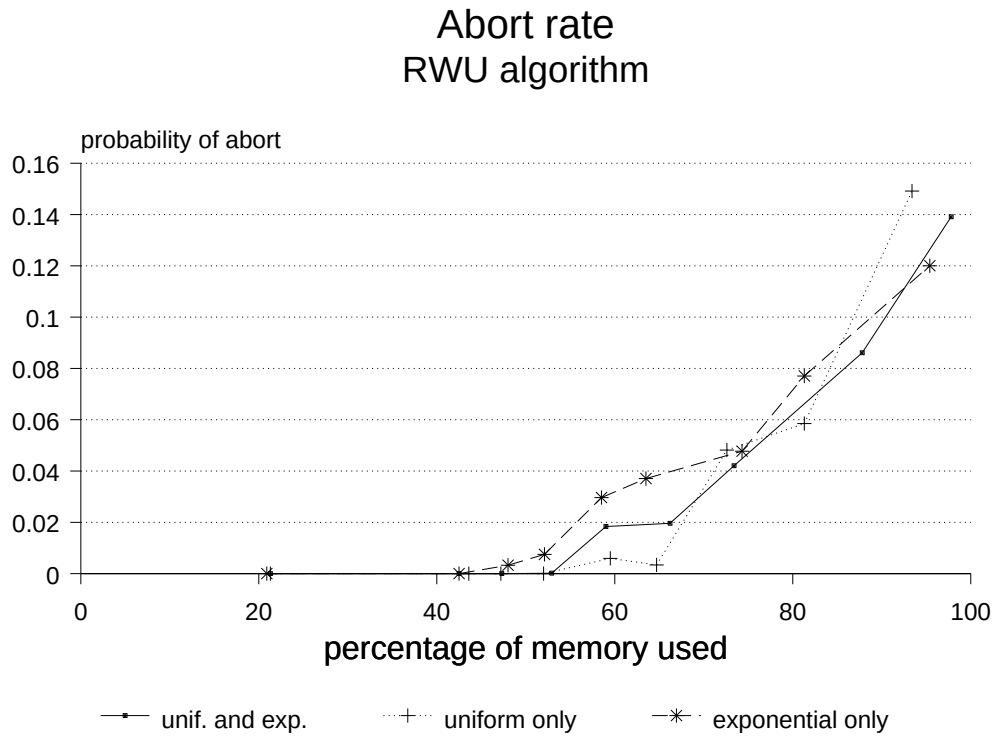

Figure 10: Impact of block size distributions on space efficiency

Appendix

A W-only Algorithm

```
allocateWonly(block,size)
int size
node **block

{
parent=anchor
Wlock(parent)
if(root is too small to allocate from)
    release locks and return(fail)
node=rightchild(parent)
Wlock(node)
while(node has a child that is larger than size)
{
    unlock(parent)
    pick child by random better-fit
    Wlock(child)
    parent=node; node=child
}
if(sizeof(node)==size)
```

```
{
    block=node
    delete(parent,node)
}
else
{
    block=getblock(node,size)
    demote(parent,node)
}
}
```

releaseWonly`(block,size)`

```
node *block
int size
{
node *parent,*child

parent=anchor
Wlock(parent)
rl=right
while(sizeof(pickchild(parent,rl))>size and not isneighbor(pickchild(parent,rl),block,size))
{ /* use pickchild to examine child information available through parent */
    child=pickchild(parent,rl)
    Wlock(child)
    unlock(parent)
    parent=child
    rl=nextchild(parent,block) /* based on address ordering*/
}
child=pickchild(parent,rl)
Wlock(child)
if(isneighbor(child,block,size))
{
    combine(parent,child,block,size)
    delete(parent,child)
/*      if child is smaller than parent, can leave in tree and use find to search for the other neighbor*/
    releaseWonly(block,size)
}
else
{
    insert(parent,child,block,size,releaselist)
    release blocks on releaselist
}
}
```

B RWU Algorithm

allocateRWU`(block,size)`

```
int size
node **block
```

```
{
parent=anchor
Rlock(parent)
if(root is too small to allocate from)
    release locks and return(fail)
node=rightchild(parent)
Rlock(node)
while(node has a child that is larger than size)
{
    unlock(parent)
    pick child by random better-fit
    Rlock(child)
    parent=node; node=child
}
unlock(node) /* unlock child to prevent deadlock */                    (2)
Upgradelock(parent) /* first W lock is an upgrade */
if(neither child is large enough)
{
    unlock(parent)
    allocateWonly(block,size) /*try again placing W locks only */
}
else
{
    pick nodeby random better fit
    Wlock(node)
    while(node has a child that is larger than size)
    {
        unlock(parent)
        pick child by random better-fit
        Wlock(child)
        parent=node; node=child
    }
    if(sizeof(node)==size)
    {
        block=node
        delete(parent,node)
    }
    else
    {
        block=getblock(node,size)
        demote(parent,node)
    }
}
}
```

releaseRWU`(block,size)`

```
node *block
int size
{
```

```
node *parent,*child

parent=anchor
Rlock(parent)
rl=right
while(sizeof(pickchild(parent,rl))>size and not isneighbor(pickchild(parent,rl),block,size))
{ /* use pickchild to examine child information available through parent */
    child=pickchild(parent,rl)
    Rlock(child)
    unlock(parent)
    parent=child
    rl=nextchild(parent,block)
}
Upgradelock(parent) /* first W lock must be an upgrade */                    (1)
while(sizeof(pickchild(parent,rl))>size and not isneighbor(pickchild(parent,rl),block,size))
{ /* find correct place to insert or delete */
    child=pickchild(parent,rl)
    Wlock(child)
    unlock(parent)
    parent=child
    rl=nextchild(parent,block)
}
child=pickchild(parent,rl)
Wlock(child)
if(isneighbor(child,block,size))
{
    combine(parent,child,block,size)
    delete(parent,child)
/*      if child is smaller than parent, can leave in tree and use find to search for the other neighbor*/
    releaseRWU(block,size)
}
else
{
    insert(parent,child,block,size,releaselist)
    release blocks on releaselist
}
}
```

C Procedures

demote(parent,child)

```
node *parent,*child
{
node *right,*left

if(isleft(child,parent)) /* test address ordering */
    rl=LEFT
else
    rl=RIGHT
```

```
righttree=rightchild(child)
lefttree=leftchild(child)
while(sizeof(righttree)>sizeof(child) or sizeof(lefttree)>sizeof(child))
{
    if(sizeof(righttree)>sizeof(lefttree))
    {
        assign(parent,rl,righttree)
        rl=LEFT
        Wlock(righttree)
        unlock(parent)
        parent=righttree
        righttree=leftchild(parent)
    }
    else
        do the same for lefttree
}
assign(parent,rl,child)
unlock(parent)
assign(child,left,lefttree)
assign(child,right,righttree)
unlock(child)
}
```

```
delete(parent,child)
node *parent,*child
{
node *left,*right

if(isleft(child,parent)) /* if the child's block is lower in memory than the parent's /*
    rl=LEFT
else
    rl=RIGHT
righttree=rightchild(child)
lefttree=leftchild(child)
while(righttree!=lefttree) /* until both are nil */
{
    if(sizeof(lefttree)>sizeof(righttree)) /*if lefttree's block is larger righttree's block */
    {
        Wlock(lefttree)
        if(isneighbor(lefttree,child,size)) /* if lefttree and child are neighbors */
        {
            temp=leftchild(lefttree) /*neighbor => no right subtree*/
            unlock(lefttree)
            combine(child,lefttree,size) /* they're neighbors, so combine into one block */
            lefttree=temp
            Wlock(lefttree)
        }
    else
    {
            assign(parent,rl,lefttree) /* make the rl (right or left) pointer of parent point to lefttree */
```

```
            unlock(parent)
            rl=RIGHT
            parent=lefttree
            lefttree=rightchild(lefttree)
        }
    }
    else
        do the same for righttree
assign(parent,rl,NULL)
unlock(parent)
}


insert(parent,child,newnode,size,releaselist)
node *newnode,*parent,*child
int size
nodelist *releaselist
{
node *child,*right,*left

if(isleft(newnode,parent))
    assign(parent,LEFT,newnode)
else
    assign(parent,RIGHT,newnode)
setsize(newnode,size)
lefthook=newnode; left1=LEFT
righthook=newnode; right1=RIGHT
Wlock(newnode,2) /* must unlock newnode twice */
unlock(parent); foundneighbor=false
while(child!=NULL and not foundneighbor)
{
    if(isleft(child,newnode))
    {
        if(isneighbor(child,newnode,size))
        {
            assign(lefthook,left1,leftchild(child))
            unlock(lefthook)
            assign(righthook,right1,rightchild(child))
            addtolist(releaselist,child) /* set aside the neighbor to be released later */
            unlock(child)
            parent=righthook
            child=rightchild(child)
            if(child!=NULL)
            {
                foundneighbor=LEFT
                lock(child)
            }
        }
        else
        {
            assign(lefthook,left1,child)
```

```
            left1=RIGHT
            unlock(lefthook)
            lefthook=child
            child=rightchild(child)
            Wlock(child)
        }
    }
    else
        do the same for righthook
}
if(foundneighbor==LEFT)
{
    while(leftchild(child)!=null)
    {
        unlock(parent)
        parent=child
        child=leftchild(child)
        lock(child)
        right1=LEFT
    }
    if(isneighbor(child,newnode,size))
    {
        assign(parent,right1,rightchild(child))
        addtolist(releaselist,child)
    }
    unlock(parent,child)
}
elseif(foundneighbor==RIGHT)
    do the same with rightchild
else
{
    assign(righthook,right1,null)
    assign(lefthook,left1,null)
    unlock(lefthook,righthook)
}
}
```

find(parent,child,sizeblock,released)

```
node *parent,*child,*block,**released
int size
{
node *pos

unlock(parent)
pos=child
released=NIL

if(child<block)
{
    rl=RIGHT
```

```
    parent=child
    child=pickchild(child,rl)
    Wlock(child)
    while(leftchild(child)!=null)
    {
        unlock(parent)
        parent=child
        child=leftchild(child)
        lock(child)
        rl=LEFT
    }
    if(isneighbor(child,pos,size))
    {
        assign(parent,rl,rightchild(child))
        released=child
    }
    unlock(parent,child)
}
else
    do the same for the left tree
}
```