A Highly Concurrent Priority Queue Based on the B-link Tree

University of Florida, Department of CIS

Electronic Tech Report #007-91

Theodore Johnson

August 18, 1991

Abstract

We present a highly concurrent priority queue algorithm based on the B-link tree, which is a B^+-tree in which every node has a pointer to its right sibling. The algorithm is built on the concurrent B-link tree algorithms proposed by Lehman and Yao [15] and Sagiv [19]. Since the priority queue is based on highly concurrent search structure algorithms, a large number of insert operations can execute concurrently with little or no interference. We present two algorithms for executing the deletemin operation. The first algorithm executes deletemin operations serially, but we show that it can support a higher throughput that previous shared memory concurrent priority queue algorithms because most deletemin operations execute very quickly. The second deletemin algorithm uses the fetch-and-add operation to allow several deletemin operations to execute concurrently, and can support a much higher throughput.

1 Introduction

A priority queue handles two operations: insert and deletemin. The *insert* operation puts a key into the queue, and the *deletemin* operation removes the smallest key from the queue and returns it. A concurrent priority queue handles many insert and deletemin requests simultaneously. The operations on a concurrent priority queue should satisfy a serializability condition such as strict serializability or decisive operation serializability [20]. Concurrent priority queues can be useful in parallel processor scheduling (especially real-time processor scheduling). In addition, concurrent priority queues are useful for a variety of parallel algorithms [17, 18, 16].

Many concurrent priority queues have been proposed, usually based on a heap structured tree. In a heap structured tree, the key in a child is greater than the key in the parent. As a result, the root must have the lowest key in the entire tree, so a deletemin operation always removes the root. Quinn and Yoo [17] describe a parallel algorithm for removing an element from a heap structured tree. Biswas and Brown [4] propose a concurrent priority queue based on a nearly complete heap-structured binary tree. Their algorithm, which allows concurrent inserts and deletemins, is a based on the usual priority

queue algorithm [1]. The authors use a special locking protocol to prevent deadlock between descending deletemin restructuring operations and ascending insert restructuring operations. Rao and Kumar [18] propose an algorithm similar to the one in [4], except that the insert operations restructure top-down. Anani [2] proposes an extension to Rao and Kumar's algorithm that evenly hashes insert operations to the external nodes. Jones [12] proposes a priority queue based on the skew heap [21],a self-adjusting heap-ordered binary tree. This algorithm admits a simple lock-coupling algorithm because both insert and deletemin operations restructure the priority queue top-down. Herlihy [8] proposes a wait-free concurrent priority queue algorithm based on the skew heap. In addition to the shared memory concurrent priority queues, Fan and Cheng [6] propose a pipelined VLSI priority queue that uses a sorting and a merging network.

An obvious problem with concurrent priority queues based on a heap-structured binary tree is that the root is a serialization bottleneck. To avoid the serialization bottleneck, we propose a concurrent priority queue based on a different data structure, the B-link tree. A B-link tree is a B^+-tree in which each node has a pointer to its right neighbor [5]. The leftmost child of a B-tree will always contain the smallest key. Thus, the leaf level of a B-link tree is an ordered list, and the internal nodes of the B-link tree form an index into the list. Many inserts of low-priority items can occur simultaneously with a deletemin operation with little or no interference.

2 The Concurrent Priority Queue

As a starting point for a concurrent priority queue algorithm based on a B-link tree we use the concurrent algorithms that have been proposed for the B-link tree [19, 15, 14]. These algorithms have been found to allow the highest throughput and to cause the least serialization delays among the existing concurrent B^+-tree algorithms [10, 9], a further indication that the B-link tree is a promising data structure. The idea behind the B-link tree algorithms is the use of the *half-split* operation (see figure 1). When an operation attempts to insert a key into a full node, it first half-splits the node, then inserts in the parent node a pointer to the newly created sibling. For a period of time, the sibling exists in the B-link tree with no corresponding entry in the parent. Operations can still navigate to the sibling during this period of time due the sibling pointers in each node. Each node stores the value of the largest key that might be found in the subtree rooted at that node. If an operation is searching for a larger key, it visits the node's right sibling.

Our concurrent priority queue algorithm builds upon Sagiv's [19], which is an improvement of the Lehman-Yao algorithm [15]. In Sagiv's algorithm, no operation holds more than one lock at a time. Search operations start by placing a shared lock on the root of the B-link tree, determining the next node to examine, and removing the lock on the root. (Sagiv didn't use shared locks, assuming instead that nodes may be read and written atomically). The search operation continues in this fashion until it finds the leaf that might contain the key it is searching for, at which point it searches that leaf, unlocks the leaf, and returns. An insert operation starts by searching for the key that it is inserting. The insert operation places an exclusive lock on the leaf, and performs the insert operation. If the leaf is too full, it half-splits the leaf, releases the lock on the leaf, locks the parent, and inserts a pointer to the sibling in the parent. If the parent is too full, the the parent is half-split and the restructuring continues until either a non-full parent node is reached or the root is split.

A priority queue requires deletemin operations in addition to the insert operations. Merging underfull nodes due to delete operations is difficult to implement in a B-link tree, because it is difficult to determine when the space used by the deleted node may be reclaimed [14, 19]. The deletemin operation has more structure than the delete operation, so that the space used by deleted nodes can be easily reclaimed. Reclaiming the unused space will require some additional data structures, which we describe next.

2.1 Data Structures

The essential data structure is the B-link tree, which is a B^+-tree in which every node contains a pointer to its right sibling (see figure 2). For the priority queue algorithm, we assume that the root node is a special node which is always the root of the tree. The initial queue consists of two nodes: the root and an empty leaf, and the root contains a pointer into the empty leaf. We assume that the root is never a leaf, so the tree has a minimum height of two.

Each node in the B-link tree contains a set of keys, $k_1, k_2, \ldots, k_m$, a *highest* field, and a set of pointers, $p_0, p_1, p_2, \ldots, p_m$ (see figure 3). In a non-leaf node, the subtree pointed to by p_i contains keys k such that $k_i < k \leq k_{i+1}$ (where $k_0 = -\infty$ and $k_{m+1} =$ `highest`). In a leaf node, pointer p_i points to the data object with key k_i. Each node also contains the pointer *rightsibling*, which points to the node's right sibling (nil if the node is the rightmost). The `highest` field in a node indicates the value of the largest key that might be stored in the subtree rooted at that node. If an operation navigates to a node and is searching for a key value larger than `highest`, the operation can follow the link `rightsibling` to find

the correct subtree.

In order to make the B-link tree concurrent, each node must contain a socket for a lock queue. Each node also contains a flag, *deleted*, which is set to true if the node is no longer in the priority queue, and a counter, *marks*, which records the number of active inserts that have read the node (the space used by a deleted node cannot be reclaimed until the `marks` counter is zero). Finally, each node will need to store which insert operation created the node in creator.

Two more data structures are needed - both lists of pointers (see figure 2). In both lists, there is one pointer for each level in the tree. In the first list, *minheadlist*, each pointer in the list points to the leftmost active (not deleted) node in a level. The pointers in `minheadlist` are ordered from the leaf level to the root level. `Minheadlist` may be locked, which implicitly locks the leftmost leaf. The deletemin operations use `minheadlist` to find the parent nodes. The other list, *delheadlist* is similar, except that its pointers point to the first node in the level, deleted or active. `Delheadlist` is maintained so that deleted nodes that will be accessed in the future can be kept in the data structure, but have their space eventually reclaimed.

2.2 Locks

This algorithm uses four types of locks, W, Wp, Wi, and R. Table 1 summarizes their compatibility. The W lock is the usual exclusive lock, and the R lock is the usual shared lock. A W lock can be upgraded to a Wp lock, which is a preemptive exclusive lock. When a W lock is upgraded to a Wp lock, all processes that have Wi (preemptable exclusive) locks relinquish their other lock to the process that placed the Wp lock. A process that attempts to place a Wi lock on a node that is Wp-locked is preempted also. When the Wp lock is released, the preempted operation resumes. The preempting and preemptable locks are used to break deadlocks. In the priority queue algorithm, the deadlock can occur at only one point, so that the Wi and Wp locks need to be implemented on only one queue (`minheadlist`). In order to make the algorithm more efficient, a Wi lock has priority over all waiting W locks.

The algorithm can be implemented without using the Wi and Wp locks, but we will first present the algorithm as using them in order to simplify the presentation.

2.3 Procedures

The insert and deletemin operations use the following operations:

	R	W	Wp	Wi
R	yes	no	no	no
W	no	no	no	no
Wp	no	no	preempted	no
Wi	no	no	no	no

Table 1: Lock Compatibility Chart

1. `mark, unmark, nummarks` : Atomically increment, decrement and read the `marks` field of a node. `Mark` and `unmark` can be implemented using the fetch-and-add operation [7].

2. `findchild(node,v,leftmost,ischild)` : Determines the next node to visit while searching for key `v`. The procedure will return the address of the right sibling if `v` isn't in the range of node, or if the node is deleted. `Leftmost` is set to false if the returned node isn't the leftmost in the node, and `ischild` is true if the returned node is a child of node, false if it is the right sibling.

3. `halfsplit` : Perform the operation in figure 1. In addition, `child` gets the address of the new sibling, and `v` gets the separator key.

4. `search` : Release the W lock on the root, search for `child` in the B-link tree (using the same search protocol as in the insert procedure), return with parents of `child` in `pstack`, and a W lock on the parent of `child`.

5. `makenewroot(root,v,child,newsib)` : Creates two new nodes, distributes the contents of the root among them, and also inserts `v` and `child`, creates entries in root for these 2 new nodes. `Newsib` returns the leftmost of the newly created nodes.

6. `addlevelptr(minheadlist,node)` : Add an entry in `minheadlist` which points to `node`.

7. `deleteminkey` : Delete and return the smallest key in node. If the node is empty, the priority queue is empty, so `deleteminkey` returns a value that indicates failure.

3 The Serialized-deletemin Algorithm

There are two priority queue procedures: insert and deletemin. The pseudo-code for these algorithms is in the Appendix. The insert procedure starts by searching the B-link tree for the leaf into which it must insert its key. The search phase of the insert algorithm is similar to the search phase in Sagiv's

algorithm, with the exception that all nodes are marked before they are visited. Nodes that have non-zero values in their marks fields are left in the data structure even after being removed from the B-link tree by the deletemin operation. Marking a node ensures that insert operations can find their paths on two occasions. The first occasion occurs during the search phase, during which the insert operation needs to ensure that the next node that it will lock actually exists in the data structure. A deletemin operation must exclusively lock the parent of a node before it can delete the child, and it must delete the left sibling of a node (and thus must place an exclusive lock on the left sibling) before it may delete a node. So, during the time period that the insert operation holds a lock on the parent node (or the left sibling) the child (or right sibling) must be in the data structure. Marking the child (or right sibling) ensures that the node is still in the tree when the insert operation places a lock on the node. The second occasion upon which marking is used is during the restructuring phase. Marks are left on the parent nodes, ensuring that they will exist if the operation needs to find parents during restructuring. Note that, as an optimization, the root doesn't need to be marked because it is never deleted.

After the insert operation navigates to a leaf, it will begin the restructuring phase, so it will place W locks. After the operation has a W lock on a node, it must consider four possibilities. First, a deletemin operation might have inserted the pointer to the child already (as will be discussed later). If this happens, the deletemin operation will notify the insert operation that its work has been finished. If an insert operation finds that it has been notified, the restructuring is finished, so the insert operation releases its locks and returns. Second, the operation might have navigated to the wrong node due to splitting or deletions. In this case, the operation follows the right sibling pointer (marking the right sibling before releasing the lock on the node). Third, the operation might be at the correct node, and the node is full. In this case, the operation half-splits the node, and navigates to the presumed parent. If the parent is the root, the operation must take some special actions. If the root height has changed (increased), the insert procedure must find prospective parents on the new levels of the B-link tree, which is accomplished by the search procedure. If the root is full, the insert operation will have to split the root and add another level. Adding a level to the B-link tree requires that `minheadlist` and `delheadlist` be modified, which requires that they be exclusively locked in order to prevent interference with deletemin operations. The insert operation places a Wi (preemptable exclusive) lock on `minheadlist` to prevent deadlock. Fourth, the insert operation might be at the correct node, and the node isn't full. In this case, the operation inserts the key and pointer, and is finished with restructuring. After completing the

restructuring, the operation unmarks all remaining marked parent nodes, and exits.

The deletemin operation begins by placing a lock on `minheadlist`, which also locks the leftmost leaf, which contains the smallest key currently in the queue. The deletemin operation then deletes the smallest key in the locked leftmost leaf. If the leaf is not empty, the deletemin operation releases its lock and returns the key. If the leaf becomes empty, it must be removed from the data structure, so the B-link tree needs to be restructured. The deletemin operation first locks all nodes that will be involved in the restructuring. The leftmost node of each level (pointed to by `minheadlist`) is exclusively locked progressing from the leaf level to the root level. If a node is the root, has more than one child, or contains an outdated entry for the locked child (indicating that the locked child has an uninserted sibling), then that node is the highest node that needs to be locked. If the root needs to be locked, the deletemin operation upgrades the `minheadlist` lock to a Wp lock, in order to break any possible deadlock. After the highest node is locked, there are three possibilities. First, the highest node might be the root, and the deletemin operation might be attempting to delete the root's only child. In this case, the deletemin operation releases all locks without modifying the data structure, as (we assume) the height of the B-link tree never decreases. Second, the node might have an outdated entry for the child, which occurs if the separator key for the child in the node is greater than the the `highest` field in the child. This situation means that the child has been split but the sibling hasn't yet been inserted into the parent node. The sibling must be inserted into the parent before the child can be deleted, because the insert operations can only move to the right, not the left. The deletemin operation must remove the pointer to the child, insert a pointer to the sibling, and notify the insert operation that created the sibling that the restructuring has been performed. Third, if the first two cases don't apply, it removes the child from the node.

Once the child is deleted (or marked as deleted) from the highest locked node, the other locked nodes are no longer reachable in the data structure, so all that remains is to perform bookkeeping on the data structures. On each locked level, the operation marks the locked node as deleted, and shifts the corresponding pointer in `minheadlist` to the right sibling. Next, it attempts to reclaim the space occupied by the deleted nodes on that level whose marks fields are zero (which means that no operation will access it again). The operation locks `delheadlist` and releases all of the other locks, so as not to block other deletemin operations needlessly. The deleted nodes are exclusively locked from left to right using lock coupling to ensure that insert operations can always read the marked nodes.

3.1 Correctness

We first show that the algorithm is deadlock-free. Insert operations place at most one lock at a time, except when splitting the root. When an insert operation splits the root, it must lock both the root and `minheadlist`. The only time that a deadlock can occur is when a deletemin operation attempts to lock the root. In this case, the deletemin will upgrade its lock on minlisthead to a preempting exclusive lock, while the insert procedure placed a preemptable exclusive lock on minlisthead, so the deadlock will be broken in favor of the deletemin operation.

We next show that a insert operation correctly inserts its key into the data structure and correctly restructures the B-link tree. If none of the nodes involved in the insert operation are deleted while the insert operation is active, then the insert operation is correct because it uses the same protocol as Sagiv's algorithm [19]. If a node that is used in the insert operation is deleted while the insert operation is active, there are two possibilities. First, the node might be deleted before the last time that the insert operation accesses it. In this case, the insert operation can always recover its path as discussed in section 3. If the node is deleted after the last time that the insert operation accesses it, then the only problem occurs when the insert operation attempts to insert a pointer to the deleted node into the node's parent. If the insert operation is trying to insert into the parent a pointer to a deleted node, then the node's left sibling must also have been deleted, which means that a deletemin operation has already inserted the node and notified the insert operation. So, the insert operation detects the notification and stops.

The deletemin operations always find the smallest key that has been inserted into a leaf because the insert operations are correct and the deletemin operations always delete the leftmost key in the data structure. The deletemin operations restructure the tree correctly because they obtain locks on all affected nodes before restructuring, and they ensure that the leftmost branch in the tree exists.. The deletemin operations never remove from the data structure any node that might be accessed in the future by an insert operation. Since the deletemin operation recovers the space used by any node that will not be read by an insert operation, the number of deleted but unrecovered nodes at any point in the algorithm is $O(I)$, where I is the maximum number of concurrent insert operations.

This algorithm is decisive operation serializable (The concurrent execution is equivalent to a serial execution in which operations are ordered according to when their *decisive actions* occurred [20]). The decisive actions of the insert and deletemin operations occurs when they actually insert a key into or

delete a key from the data structure.

3.2 Implementing Notification

We have not yet specified a method by which deletemin operations can notify insert operations that their restructuring has been completed. In this section, we suggest three methods that do not rely on system services, each of which may be appropriate in different situations.

The simplest method is to use a bit vector, with one bit representing a particular process that issues insert operations. Each insert operation reads the bit that corresponds to the process that is executing the operation when the operation starts. Whenever an operation half-splits a node, it writes its process number in the `creator` field of the newly created sibling. Whenever a deletemin operation inserts a sibling into a parent, it negates the bit that corresponds to the sibling's creator. Exery time that an insert operation obtains a lock during the restructuring phase, it tests the bit that corresponds it its process number. If the bit has changed, a deletemin operation has notified the insert operation that the restructuring is finished. Since only one process writes to the bit vector at a time, and since the insert operation that will read the bit being changed by the deletemin is blocked by the deletemin's lock on the parent node, no synchronization needs to be performed on the bit vector. This method is simple and efficient, but can only be used when the processes performing insert operations can be efficiently globally numbered in advance.

A slightly more complicated method is to have the insert operations request numbers for use with a bit vector. Only the insert operations that split a node must request a number. This method allows an arbitrary number of different processes to perform insert operations on the queue, but requires that they synchronize to obtain numbers, and maintain a free list so that the numbers can be reused, which requires much more space than the bit vector.

A third method is to maintain the notifications within the priority queue. In this method, whenever a deletemin operation inserts a sibling into a parent, it attaches a notification to the parent (using a queue of notifications). If an insert operation finds a notification corresponding to its process number in the parent's notification queue, it removes the notification and exits. In order to guarantee that an insert operation finds its notification, the notification must always be in the node that is or would be the inserted sibling's parent (so the notification must indicate the inserted sibling's key range). This requires that the notification queue must be split whenever a node half-split. Whenever the root is split, its notification

queue must be split among the two new children. Whenever a node is deleted, the notification list must be appended to the right sibling's notification list. This method allows arbitrary numbers of insert operations to execute simultaneously, but at the cost of greatly complicating the implementation.

3.3 Using Read and Write Locks Only

While one can expect that the underlying system will support the common R and W locks, the Wi and Wp lock is not likely to be provided by the system. In this case, one can either implement the deadlock-breaking locks, or modify the algorithm so that it avoids deadlock. In this section, we describe how the algorithm can become deadlock-avoiding at the cost of a slightly more complicated algorithm

Deadlock in the priority queue algorithm is possible because the insert operation will lock the root and then `minheadlist`, while the deletemin operation locks `minheadlist` and then the root. In order to break the lock, we require that the if the insert operation needs both locks simultaneously, `minheadlist` is locked before the root.

We could satisfy this requirement by forcing any insert operation which locks the root to lock `minheadlist` first. While this solution has the advantage of simplicity (only one `if` statement needed), `minheadlist` would be locked (and thus the deletemin operations blocked) more often than necessary. A higher concurrency solution is to require that any insert operation that locks the root and determines that it needs to split the root first release the lock on the root and then lock `minheadlist` and the root, in that order. In either case, the root height might have changed, but this possibility has already been covered with the use of the `search` procedure. Again, in either case, the root should be split only if it is full.

3.4 Decreasing the Tree Height

As stated, the height of the priority queue never decreases. This is probably not a serious problem in practice, since creating a height h tree requires an exponentially large number of insert operations. However, a priority queue might start out large, then shrink to some much smaller size and remain stable at that size, so that a priority queue that can't decrease in height will cause considerable wasted work. The algorithm can be modified so that the tree will shrink, at the cost of a slightly more complicated implementation.

When the result of a deletemin operation is a root with one child and a height greater than two, the

tree can be shrunk by copying the contents of the child of the root into the root. The remaining child of the root is the right sibling of the deleted child, and must be W-locked so that it can be removed from the tree. If the remaining child has a non-null right sibling, then it has been split and there is another child of the root. In that case, insert the right sibling into the root, and notify the insert operation that created it. Otherwise, copy the contents of the remining child into the root and mark the child as deleted.

We must now ensure that insert operations that navigate to a deleted level (whether during the search or restructuring phase) can find their way back into the tree. Navigation to a deleted level will be detected when the insert operation attempts to navigate to a right sibling and finds that the pointer is null. In this case, the operation must renavigate starting from the root.

In order to reclaim the memory used by the nodes on the deleleted level, the chain of nodes pointed to by the level's entry in `delheadlist` should become an entry in a data structure similar to `delheadlist`, *deletedlevellist*. This data structure must be searched (as `delheadlist` is) in order to return unmarked nodes to free memory.

If the third option for notification is used (keeping the notifications in the tree), then the notification list of the the child should be appended to the root's list.

4 The Concurrent-Deletemin Algorithm

A priority queue algorithm that allows concurrent deletemin operations can be implemented if the fetch-and-add operation [7] is available. The idea is that deletemin operations can concurrently perform a fetch-and-add on the number of remaining items in the leftmost node. Each deletemin operation receives a different number which corresponds to the item in the leftmost node which it returns. Up to m deletemin operations can execute concurrently by using this scheme (where m is the maximum number of keys that can be stored in a node).

A simple modification of the serial-deletemin algorithm is to remove the leftmost leaf from the tree and store it in a block that only the deletemin operations can access. A deletemin operation performs a fetch-and-decrement (fetch-and-add with an argument of -1) on the shared variable *numleft*. If the returned value is greater then zero, the key to remove from the block is given by the original number of keys in the block minus the value returned by the fetch-and-decrement. If the fetch-and-decrement returns 0, then the deletemin operation is the first to discover that the node is empty, so it is responsible

for removing the current leftmode leaf from the tree and copying it into the block. The last action of this deletemin operation is to set `numleft` to the number of keys in the removed node. If the fetch-and-decrement instruction returns a negative value, the block is empty and the deletemin operation must try again.

This implementation is simple and efficient, but it has the problem that it is can produce executions that aren't strict serializable [20] – that is, it is possible for that an insert operation I finishes before deletemin operation D starts, I inserts k_I, D deletes k_D, $k_I < k_D$, and k_I is still in the priority queue when D finishes. The non-strict serializable implementation may be acceptable, especially if there is a high rate of deletemin operations and inserts into the leftmost leaf are rare (i.e., keys of the insert operations tend to increase), but some applications might require a strict serializable or a decisive operation serializable [20] priority queue.

We next present a decisive operation serializable (and thus a strict serializable) concurrent-deletemin priority queue. In order to be decisive operation serializable, after an insert puts its key into the tree (i.e. performs its decisive operation), all deletemin operations whose decisive operations follow must be able to delete this key. Therefore, insert operations must be able to insert into the block that the delete operations delete from.

In the decisive operation serializable priority queue, there is now a separate lock for minheadlist and for the leftmost block in the tree. A deletemin operation places an R lock on minheadlist and reads the pointer to the leftmost leaf. The deletemin operation then marks the leafmost leaf, releases the lock on minheadlist, and R-locks the leftmost leaf. If the leftmost leaf isn't deleted, the deletemin operation performs a fetch-and-decrement on numleft. If the returned value is positive, the deletemin deletes a key, releases the lock, and unmarks the node. If the value returned by the fetch-and-decrement is negative, the operation releases the lock, unmarks the node, and waits until the leftmost node is removed from the tree, which the operation can detect by monitoring the pointer to the leftmost node. When this pointer changes, the operation starts again. If the returned value is zero, then the operation is responsible for removing the empty leaf and finding the next leftmost leaf. This operation releases its lock, places a W lock on minheadlist and the leftmost leaf, then proceeds as in the serialized-deletemin algorithm.

When an insert operation needs to insert its key into the leftmost leaf, it places a W lock on the leftmost leaf. If the insert operation finds the that the leftmost leaf isn't empty, it inserts its key and modifies the value of numleft and of the field of the node that holds the number of keys in the node, and

possibly half-splits the node. Otherwise, the insert operation follows the right sibling pointer.

5 Analysis and Comparisons

In this section, we will analyze the maximum throughput of the serialized-deletemin and the concurrent-deletemin algorithms and use the analysis to compare them to the existing concurrent priority queue algorithms.

In order to analyze the algorithms, we need to make some assumptions about the workload offered to the algorithms. First, we assume that the keys of the insert operations tend to increase This assumption holds for many applications, such as discrete event simulators, and is the working assumption made when empirically comparing priority queue algorithms [11, 13]. Second, we assume that the rate of insert operations is equal to the rate of deletemin operations, so that the size of the priority queue is stable.

5.1 Analysis of the Serialized-deletemin Algorithm

In order to analyze the algorithms, we need to characterize the data structure. Baeza-Yates [3] has shown that the expected utilization of a B^+-tree is 69% if the tree is created from a sequence of random insertions. Since the deletemin operations delete from only one node at a time, we will assume that the expected space utilization of the nodes in the tree (except for the leftmost leaf) is 69%.

We first consider the insert operations. Since we assume that the keys of the insert operations tend to increase, there is little interference between the insert operations and the deletemin operations. Therefore, it is primarily insert operations that block other insert operations. The insert operations follow the Sagiv's protocol, which Johnson and Shasha have shown results in very little interference [10, 9]. Since we assume that the rate of the insert operations equals the rate of the deletemin operations, the throughput of the priority queue will be limited by the maximum rate at which deletemin operations can be processed.

In the serialized-deletemin priority queue, the deletemin operations locks minheadlist (and thus implicitly the leftmost leaf), and if the leftmost leaf has more than one key in it, finishes and releases its lock. If the deletemin operation deletes the last key in the leftmost leaf, it locks the parent and removes the pointer to the child. If the parent now becomes empty, the grandparent must be locked, and so on.

Let us denote the time to (successfully) lock minheadlist, delete the smallest key, and release the lock by D. Let us denote the additional time to restructure one level of the tree by R. Let M represent the maximum number of keys that can be stored in a node, and let $E = .69M$. Let T_D be the expected time

to execute a deletemin operation

Since we've assumed that insert operations rarely conflict with deletemin operations, the limiting source of blocking is between deletemin operations. Since the deletemin operations are (almost) serialized, the maximum throughput of the deletemin operations is the inverse of their expected service time. Every deletemin operation requires D seconds. If the leaf has only one key in it, an additional R seconds are required. Since inserts rarely insert into the leftmost leaf, this happens once every E deletemin operations. If the leftmost leaf must be removed, the parent might also become empty, and we'll assume that again this happens once every E deletions from the parent of the leftmost leaf. Continuing this way for a tree of arbitrary height we find that

$$\begin{aligned} T_d \approx & \quad D + R\sum_{i=1}^{\infty}(1/E)^i \\ = & \quad D + R/(E-1) \\ \approx & \quad D \end{aligned}$$

Therefore, the throughput of the serialized-deletemin is approximately $2/D$.

5.2 Analysis of the Concurrent-Deletemin Priority Queue

For this analysis, we will make the same assumptions as for the serialized-deletemin queue. So, again the emphasis is on determining the throughput of the deletemin operations, and in particular the service time of the serializing operation.

In the concurrent-deletemin queue, the deletemin operations work in parallel as long as there is a key to delete from the leftmost leaf. So, the deletemin operations are serialized by the deletemin operations which are responsible for restructuring the tree.

Let S be the time required to place a R lock and read a variable. The time to execute a deletemin operation that restructures the tree is the time to R-lock and read minheadlist (S), R-lock and delete a key from the leftmost leaf (D), W-lock minheadlist and the leftmost leaf ($L_1 + L_2$), and restructure the tree ($ER/(E-1)$). All of these times are known except for L_1 and L_2, which are the times to place a W lock on a node given that there is a stream on R locks being placed on the node and no other W locks are being placed. This type of queue is analyzed in [9], where it is shown that

$$L \approx (ln(\lambda_R/\mu_R) + \gamma)/\mu_R$$

where L is the time to place a W lock, $\gamma \approx .5721$ is Euler's constant, R locks arrive at rate λ_R and are served at rate μ_R. In this case, $\lambda_R = \lambda$, the rate at which deletemin operations arrive, and $1/\mu_R = S$. So,

$$L_1 + L_2 = 2S(\ln(S\lambda) + \gamma))$$

Therefore,

$$T_d \approx S + D + ER/(E-1) + 2S(\ln(S\lambda) + \gamma)$$

The maximum rate at which deletemin operations can be served is bound by the rate at which restructuring deletemin operations arrive so that they are always executing. One of every E deletemin operations restructures the tree, so λ is bound by the solution of

$$(S + D + ER/(E-1) + 2S(\ln(S\lambda) + \gamma))\lambda/E = 1$$

Since λ is certainly less than $E/S + D + ER/(E-1) + 2S\gamma$, we can conclude that

$$\begin{aligned} \lambda &\approx \frac{E}{S+D+ER/(E-1)+2S(\ln(S(E/S+D+ER/(E-1)+2S\gamma)+\gamma))} \\ &\approx \frac{E}{D+R} \end{aligned}$$

Therefore, the throughput of the concurrent-deletemin priority queue is approximately $2E/(D+R)$.

5.3 Comparison

Both the serialized-deletemin and the concurrent-deletemin B-link tree based concurrent priority queue algorithms are superior to the shared memory priority queue algorithms that have appeared in the literature to date. Fan and Cheng's algorithm [6] can provide better performance, but it is a synchronous VLSI implementation, so it isn't directly comparable.

The Biswas-Browne priority queue algorithm [4] serializes deletemin operations at the root, and requires that during the serialization period, three locks be placed (including one on the root) and that the root restructured on every deletemin operation. The Rao-Kumar algorithm [18] (improved upon by Anani [2]) serializes both inserts and deletemins at the root, requires that four locks be placed (including one on the root), and that the root be restructured on every deletemin operation. Jones' algorithm [12] serializes insert and deletemin operations at the root, and requires that the deletemin operations place two locks, including one on the root, and and that the root be restructured.

All of the above algorithms allow many deletemin operations to be active in the queue concurrently. Our serialized-deletemin algorithm, by contrast, allows at most two (and usually one) deletemin operation to be active at any given time. However, concurrent data structures should not be measured by their concurrency, instead they should be measured by their throughput and response times. By these measures, the serialized-deletemin algorithm is superior than previous shared memory algorithms.

The serialized-deletemin algorithm will have a higher maximum throughput for the deletemin operations than previous algorithms because the serialization period is shorter: For $E-1$ out of E deletemin operations, only one lock needs to be placed (which, like the other algorithms, is equivalent to a semaphore or a busy-wait), and no restructuring needs to be performed (it can be left for the insert operations). For $E-1$ out of E of the remaining deletemin operations, only 2 locks need to be placed, and only one node needs to be restructured, and so on. On average, less time is required to complete the serializing work. Further, there is little contention between the insert and deletemin operations, whereas three of the four of the previously proposed algorithms serialize insert operations on the root, and in the fourth (Biswas-Browne) insert and deletemin operations serialize on the last node in the tree. The serialized-deletemin algorithm will also have a higher maximum throughput for the insert operations, because the insert operations will have little conflict with each other or with the deletemin operations (as previously discussed), whereas in three out of four of the previous algorithms, insert operations serialize at the root, and in the fourth (Biswas-Browne), conflict is likely near the leaves.

We have established that the serialized deletemin algorithm will cause fewer serialization delays than the algorithms previously discussed. The serialized-delete algorithm will have a faster response time even in the absence of contention, because far less locking is required. Even for relatively small nodes (say, $E = 10$), a relatively large tree will be short (10,000 items will require 4 levels), so that only a few locks need to be placed. Also, the insert operations will rarely need to restructure the tree, and most deletemin operations will be very fast. Thus, the proposed algorithms are more efficient than those previously discussed.

The concurrent-deletemin algorithm will allow an even higher maximum throughput for the deletemin operations than the serialized-deletemin algorithm. Since the maximum throughput of the deletemin operations is proportional to the E, this algorithm has the property that the data structure can be modified to allow a higher throughput. If a series of problems is presented that requires the processing of more data, so that the expected size of the priority queue is larger, the average node size can be increased

in order to allow more parallelism. For example, E can grow as $|PQ|^{1/4}$ (where $|PQ|$ is the expected size of the queue) and still maintain a tree of height 4.

5.4 Node Size

As can be seen from the previous discussion, the performance of the proposed algorithms tends to increase as the node size increases. This trend can't continue indefinitely, since if a single node can contain the entire queue, the algorithm reduces to maintaining a sorted list. The nodes should be small enough that many are required to store the queue, so that the insert operations will be hashed out among them. For a rule of thumb, plan on a four level tree, so set $E = |PQ|^{1/4}$, or a node size of $M = E/.69 = |PQ|^{1/4}/.69$.

6 Conclusions

We have presented a highly concurrent priority queue based on the B-link tree. In it, a large number of insert operations can execute concurrently with a deletemin operation and with little or no interference between operations. We discussed two algorithms for implementing the deletemin operation. In the first algorithm, deletemin operations are serialized with respect to each other, but most deletemin operations finish very quickly. The second deletemin algorithm uses the fetch-and-add instruction to allow up to M deletemin operations to execute concurrently, where M is the maximum number of keys that can be stored in a node.

An analysis of the algorithms shows that they are interesting for both theoretical and pragmatic reasons. Of theoretical interest is the fact that the concurrent-deletemin algorithm is the first shared memory priority queue algorithm that allows completely concurrent execution of at least some of the deletemin operations – all previous shared memory algorithms serialized the deletemin operations by contention for the root of the tree. The algorithms are also of pragmatic interest, because they require far less locking and restructuring that previously proposed algorithms, which shows that even the serialized-deletemin algorithm will have higher performance (measured by response time or maximum throughput) than the previously proposed algorithms.

7 Acknowledgement

I'd like to thank Dennis Shasha for his helpful comments concerning concurrent priority queues.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[2] R. Anani. Lr-algorithm: Concurrent operations on priority queues. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 22–25, 1990.

[3] R. Baeza-Yates. Expected behavior of B^+-trees under random inserts. *Acta Informatica*, 27, 1989.

[4] J. Biswas and J.C. Browne. Simultaneous update of priority structures. In *Proceedings of the International Conference on Parallel Processing*, pages 124–131, 1987.

[5] D. Comer. The ubiquitious B-tree. *ACM Comp. Surveys*, 11:121–137, 1979.

[6] A. Fan and K. Cheng. A simultaneous access priority queue. In *Proceedings of the international Conference on Parallel Processing*, pages I95–I99, 1989.

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

[8] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceeding of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206. ACM, 1989.

[9] T. Johnson. *The Performance of Concurrent Data Structure Algorithms.* PhD thesis, NYU Dept. of Computer Science, 1990.

[10] T. Johnson and D. Shasha. A framework for the performance analysis of concurrent B-tree algorithms. In *ACM SIGACT/SIGMOD/SIGART Symposium on Principles of Database Systems*, pages 273–287, 1990.

[11] D. Jones. An emperical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, 1986.

[12] D. Jones. Concurrent operations on priority queues. *Communications of the ACM*, 32(1):132–137, 1989.

[13] J.H. Kingston. Analysis of tree algorithms for the simulation event list. *Acta Informatica*, pages 15–33, 1985.

[14] V. Lanin and D. Shasha. A symmetric concurrent B-tree algorithm. In *1986 Fall Joint Computer Conference*, pages 380–389, 1986.

[15] P.L. Lehman and S.B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[16] M. Quinn and N. Deo. Parallel graph algorithms. *Computing Surveys*, 16(3):319–348, 1984.

[17] M. Quinn and Y. Yoo. Data structures for the efficient solution of graph theoretic problems on tightly-coupled computers. In *Proceedings of the International Conference on Parallel Processing*, pages 431–438, 1984.

[18] V. Rao and V. Kumar. Concurrent access of priority queues. *IEEE Transactions on Computers*, 37(12):1657–1665, 1988.

[19] Y. Sagiv. Concurrent operations on B^*-trees with overtaking. In *Fourth Annual ACM SIGACT/SIGMOD Symposium on the Principles of Database Systems*, pages 28–37. ACM, 1985.

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

[21] D. Sleator and R. Tarjan. Self adjusting heaps. *Siam Journal of Computing*, 15(1):52–59, 1986.

Appendix

```
insert(v : key)
{
    node=root
    initialize(pstack) /*create empty stack of parents*/
    Rlock(node)
    localheight=heightof(root) /* record the current root height */
    mark(node)
    leftmost=true
    child=findchild(node,v,leftmost,ischild)
    push(node,pstack) /* record the root as a parent */
    while not isleaf(child) do
    {
        mark(child)
        unlock(node)
        Rlock(child)
        if(ischild)
            push(node,pstack) /*record the node as a parent for restructuring */
        else
            unmark(node)
        node=child
        child=findchild(node,v,leftmost,ischild)
    }
    mark(child)
    unlock(node)
    if leftmost
        Wlock(minlisthead)
    else
        Wlock(child)
    node=child; child=nil; continue=true
    while(continue)
    {
        if(notified()) /*Has the insert already been performed? */
        {
            unlock(node)
            unmark(node)
            continue=false
        }
        elseif(isdeleted(node) or not inrange(node,v)) /*need to go to right sibling*/
        {
            nextnode=rightsibling(node)
            mark(nextnode)
            unlock(node,leftmost) /*unlocks minlisthead if node is the leftmost child
            unmark(node)
            node=nextnode
            Wlock(node)
        }
        elseif(isfull(node))
        {
            if(node==root)
```

```
            {
                Wilock(minheadlist)
                Wlock(delheadlist)
                if(not notified())
                {
                    makenewroot(root,v,child,newsib)
                    addptr(minheadlist,delheadlist,newsib)
                }
                unlock(root); unlock(minheadlist)
                continue=false
            }
            else
            {
                halfsplit(node,child,v)
                unlock(node,leftmost)
                unmark(node)
                node=pop(pstack)
                Wlock(node)
                if(node==root and localheight<>heightof(root))
                        search(child,pstack,node) /*if the root height changed, find more parents */
                }
            }
        else
        {
            insertkey(node,child,v)
            unlock(node)
            unmark(node)
            continue=false
        }
    }
    while(not isempty(pstack)) /*unmark remining ancestors*/
    {
        node=pop(pstack)
        unmark(node)
    }
}
```

```
deletemin() : key
{
    Wlock(minheadlist)
    minhead=headof(minheadlist) /*get first entry on minheadlist (points to leftmost leaf/*
    deleteminkey(minhead->node,v)
    if(sizeof(minhead->node)==0) /*if node became empty, need to restructure */
    {
        child=minhead->node
        minhead=next(minhead) /* get the next entry on minheadlist*/
        node=minhead->node
        Wlock(node)
        while(sizeof(node)==1 and and node<>root and not missingsibling(node,child))do
        {
            minhead=next(minhead)
            child=node
            node=minhead->node
            if(node==root)
                Wplock(minheadlist)
            Wlock(node)
        }
        if(node==root and sizeof(root)==1)
            release all locks
        else
        {
            if(missingsibling(node,child)
            {
                Rlock(rightsibling(child))
                substitute(node,child,rightsibling(child))
                notify(creatorof(rightsibling(child)))
            }
            else
                remove(node,child)
            unlock(node)
            mintemp=headof(minheadlist)
            while mintemp->node<>node do
            {
                setdeleted(mintemp->node)
                unlock(mintemp->node)
                assign(mintemp,rightsibling(node))
            }
            lock(delheadlist)
            release all other locks
            cleanup(delheadlist)
            unlock(minheadlist)
        }
    }
    else
        unlock(minheadlist)
    return(v)
}
```

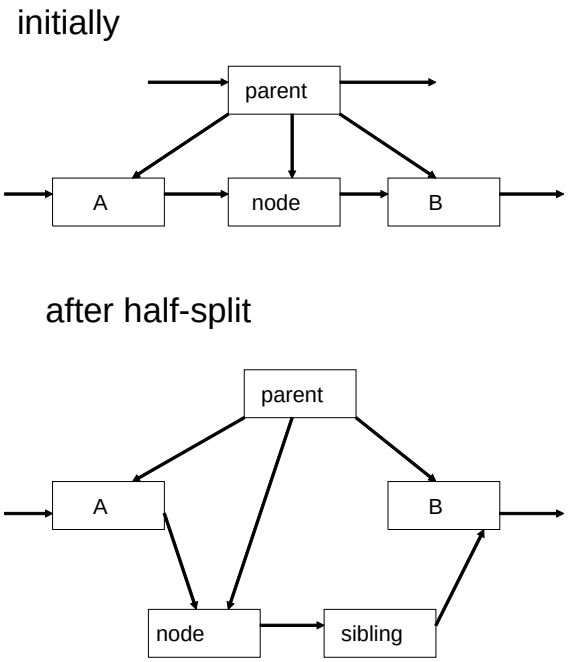

Figure 1: The half-split operation

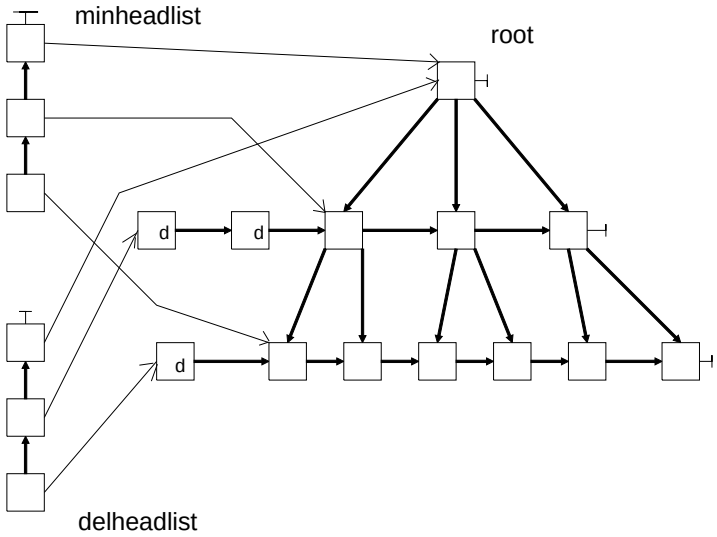

Figure 2: Priority queue data structures

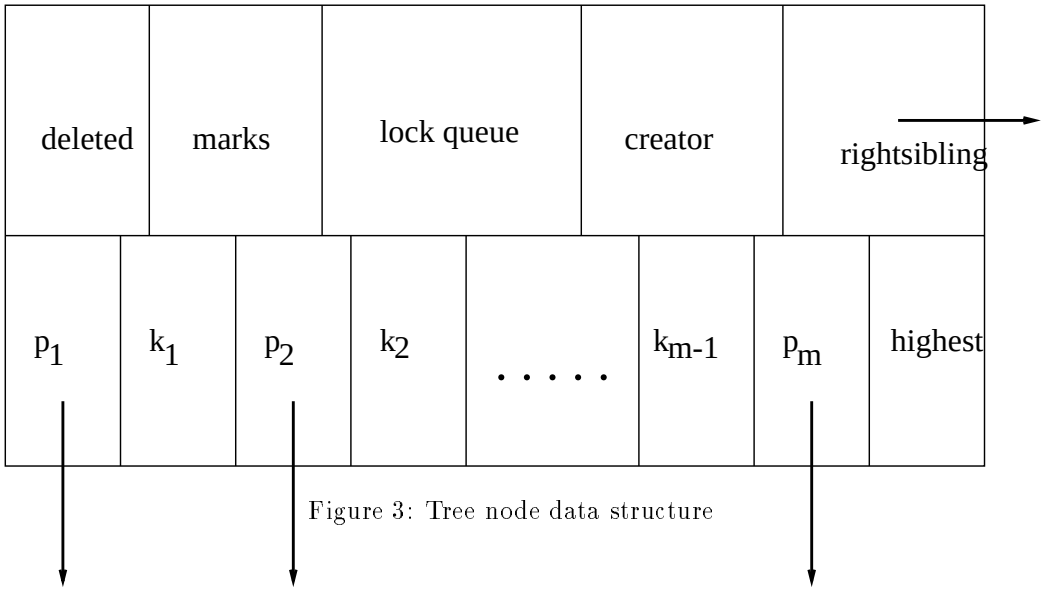

Figure 3: Tree node data structure