XArray

Author: Matthew Wilcox

Overview

The XArray is an abstract data type which behaves like a very large array of pointers. It meets many of the same needs as a hash or a conventional resizable array. Unlike a hash, it allows you to sensibly go to the next or previous entry in a cache-efficient manner. In contrast to a resizable array, there is no need to copy data or change MMU mappings in order to grow the array. It is more memory-efficient, parallelisable and cache friendly than a doubly-linked list. It takes advantage of RCU to perform lookups without locking.

The XArray implementation is efficient when the indices used are densely clustered; hashing the object and using the hash as the index will not perform well. The XArray is optimised for small indices, but still has good performance with large indices. If your index can be larger than `ULONG_MAX` then the XArray is not the data type for you. The most important user of the XArray is the page cache.

Normal pointers may be stored in the XArray directly. They must be 4-byte aligned, which is true for any pointer returned from kmalloc() and alloc_page(). It isn't true for arbitrary user-space pointers, nor for function pointers. You can store pointers to statically allocated objects, as long as those objects have an alignment of at least 4.

You can also store integers between 0 and `LONG_MAX` in the XArray. You must first convert it into an entry using xa_mk_value(). When you retrieve an entry from the XArray, you can check whether it is a value entry by calling xa_is_value(), and convert it back to an integer by calling xa_to_value().

Some users want to tag the pointers they store in the XArray. You can call xa_tag_pointer() to create an entry with a tag, xa_untag_pointer() to turn a tagged entry back into an untagged pointer and xa_pointer_tag() to retrieve the tag of an entry. Tagged pointers use the same bits that are used to distinguish value entries from normal pointers, so you must decide whether they want to store value entries or tagged pointers in any particular XArray.

The XArray does not support storing IS_ERR() pointers as some conflict with value entries or internal entries.

An unusual feature of the XArray is the ability to create entries which occupy a range of indices. Once stored to, looking up any index in the range will return the same entry as looking up any other index in the range. Storing to any index will store to all of them. Multi-index entries can be explicitly split into smaller entries, or storing `NULL` into any entry will cause the XArray to forget about the range.

Normal API

Start by initialising an XArray, either with DEFINE_XARRAY() for statically allocated XArrays or xa_init() for dynamically allocated ones. A freshly-initialised XArray contains a `NULL` pointer at every index.

You can then set entries using xa_store() and get entries using xa_load(). xa_store will overwrite any entry with the new entry and return the previous entry stored at that index. You can use xa_erase() instead of calling xa_store() with a `NULL` entry. There is no difference between an entry that has never been stored to, one that has been erased and one that has most recently had `NULL` stored to it.

You can conditionally replace an entry at an index by using xa_cmpxchg(). Like cmpxchg(), it will only succeed if the entry at that index has the 'old' value. It also returns the entry which was at that index; if it returns the same entry which was passed as 'old', then xa_cmpxchg() succeeded.

If you want to only store a new entry to an index if the current entry at that index is `NULL`, you can use xa_insert() which returns `-EBUSY` if the entry is not empty.

You can copy entries out of the XArray into a plain array by calling xa_extract(). Or you can iterate over the present entries in the XArray by calling xa_for_each(), xa_for_each_start() or xa_for_each_range(). You may prefer to use xa_find() or xa_find_after() to move to the next present entry in the XArray.

Calling xa_store_range() stores the same entry in a range of indices. If you do this, some of the other operations will behave in a slightly odd way. For example, marking the entry at one index may result in the entry being marked at some, but not all of the other indices. Storing into one index may result in the entry retrieved by some, but not all of the other indices changing.

Sometimes you need to ensure that a subsequent call to xa_store() will not need to allocate memory. The xa_reserve() function will store a reserved entry at the indicated index. Users of the normal API will see this entry as containing `NULL`. If you do not need to use the reserved entry, you can call xa_release() to remove the unused entry. If another user has stored to the entry in the meantime, xa_release() will do nothing; if instead you want the entry to become `NULL`, you should use xa_erase(). Using xa_insert() on a reserved entry will fail.

If all entries in the array are `NULL`, the xa_empty() function will return `true`.

Finally, you can remove all entries from an XArray by calling xa_destroy(). If the XArray entries are pointers, you may wish to free the entries first. You can do this by iterating over all present entries in the XArray using the xa_for_each() iterator.

Search Marks

Each entry in the array has three bits associated with it called marks. Each mark may be set or cleared independently of the others. You can iterate over marked entries by using the xa_for_each_marked() iterator.

You can enquire whether a mark is set on an entry by using xa_get_mark(). If the entry is not `NULL`, you can set a mark on it by using xa_set_mark() and remove the mark from an entry by calling xa_clear_mark(). You can ask whether any entry in the XArray has a particular mark set by calling xa_marked(). Erasing an entry from the XArray causes all marks associated with that entry to be cleared.

Setting or clearing a mark on any index of a multi-index entry will affect all indices covered by that entry. Querying the mark on any index will return the same result.

There is no way to iterate over entries which are not marked; the data structure does not allow this to be implemented efficiently. There are not currently iterators to search for logical combinations of bits (eg iterate over all entries which have both `XA_MARK_1` and `XA_MARK_2` set, or iterate over all entries which have `XA_MARK_0` or `XA_MARK_2` set). It would be possible to add these if a user arises.

Allocating XArrays

If you use DEFINE_XARRAY_ALLOC() to define the XArray, or initialise it by passing `XA_FLAGS_ALLOC` to xa_init_flags(), the XArray changes to track whether entries are in use or not.

You can call xa_alloc() to store the entry at an unused index in the XArray. If you need to modify the array from interrupt context, you can use xa_alloc_bh() or xa_alloc_irq() to disable interrupts while allocating the ID.

Using xa_store(), xa_cmpxchg() or xa_insert() will also mark the entry as being allocated. Unlike a normal XArray, storing `NULL` will mark the entry as being in use, like xa_reserve(). To free an entry, use xa_erase() (or xa_release() if you only want to free the entry if it's `NULL`).

By default, the lowest free entry is allocated starting from 0. If you want to allocate entries starting at 1, it is more efficient to use DEFINE_XARRAY_ALLOC1() or `XA_FLAGS_ALLOC1`. If you want to allocate IDs up to a maximum, then wrap back around to the lowest free ID, you can use xa_alloc_cyclic().

You cannot use `XA_MARK_0` with an allocating XArray as this mark is used to track whether an entry is free or not. The other marks are available for your use.

Memory allocation

The xa_store(), xa_cmpxchg(), xa_alloc(), xa_reserve() and xa_insert() functions take a gfp_t parameter in case the XArray needs to allocate memory to store this entry. If the entry is being deleted, no memory allocation needs to be performed, and the GFP flags specified will be ignored.

It is possible for no memory to be allocatable, particularly if you pass a restrictive set of GFP flags. In that case, the functions return a special value which can be turned into an errno using xa_err(). If you don't need to know exactly which error occurred, using xa_is_err() is slightly more efficient.

Locking

When using the Normal API, you do not have to worry about locking. The XArray uses RCU and an internal spinlock to synchronise access:

No lock needed:

- xa_empty()
- xa_marked()

Takes RCU read lock:

- xa_load()
- xa_for_each()
- xa_for_each_start()
- xa_for_each_range()
- xa_find()
- xa_find_after()
- xa_extract()
- xa_get_mark()

Takes xa_lock internally:

- xa_store()
- xa_store_bh()
- xa_store_irq()
- xa_insert()
- xa_insert_bh()
- xa_insert_irq()
- xa_erase()
- xa_erase_bh()
- xa_erase_irq()

- xa_cmpxchg()
- xa_cmpxchg_bh()
- xa_cmpxchg_irq()
- xa_store_range()
- xa_alloc()
- xa_alloc_bh()
- xa_alloc_irq()
- xa_reserve()
- xa_reserve_bh()
- xa_reserve_irq()
- xa_destroy()
- xa_set_mark()
- xa_clear_mark()

Assumes xa_lock held on entry:

- __xa_store()
- __xa_insert()
- __xa_erase()
- __xa_cmpxchg()
- __xa_alloc()
- __xa_set_mark()
- __xa_clear_mark()

If you want to take advantage of the lock to protect the data structures that you are storing in the XArray, you can call xa_lock() before calling xa_load(), then take a reference count on the object you have found before calling xa_unlock(). This will prevent stores from removing the object from the array between looking up the object and incrementing the refcount. You can also use RCU to avoid dereferencing freed memory, but an explanation of that is beyond the scope of this document.

The XArray does not disable interrupts or softirqs while modifying the array. It is safe to read the XArray from interrupt or softirq context as the RCU lock provides enough protection.

If, for example, you want to store entries in the XArray in process context and then erase them in softirq context, you can do that this way:

```
void foo_init(struct foo *foo)
{
    xa_init_flags(&foo->array, XA_FLAGS_LOCK_BH);
}

int foo_store(struct foo *foo, unsigned long index, void *entry)
{
    int err;

    xa_lock_bh(&foo->array);
    err = xa_err(__xa_store(&foo->array, index, entry, GFP_KERNEL));
    if (!err)
        foo->count++;
    xa_unlock_bh(&foo->array);
    return err;
}

/* foo_erase() is only called from softirq context */
void foo_erase(struct foo *foo, unsigned long index)
{
    xa_lock(&foo->array);
    __xa_erase(&foo->array, index);
    foo->count--;
    xa_unlock(&foo->array);
}
```

If you are going to modify the XArray from interrupt or softirq context, you need to initialise the array using xa_init_flags(), passing `XA_FLAGS_LOCK_IRQ` or `XA_FLAGS_LOCK_BH`.

The above example also shows a common pattern of wanting to extend the coverage of the xa_lock on the store side to protect some statistics associated with the array.

Sharing the XArray with interrupt context is also possible, either using xa_lock_irqsave() in both the interrupt handler and process context, or xa_lock_irq() in process context and xa_lock() in the interrupt handler. Some of the more common patterns have helper functions such as xa_store_bh(), xa_store_irq(), xa_erase_bh(), xa_erase_irq(), xa_cmpxchg_bh() and xa_cmpxchg_irq().

Sometimes you need to protect access to the XArray with a mutex because that lock sits above another mutex in the locking hierarchy. That does not entitle you to use functions like __xa_erase() without taking the xa_lock; the xa_lock is used for lockdep validation and will be used for other purposes in the future.

The __xa_set_mark() and __xa_clear_mark() functions are also available for situations where you look up an entry and want to atomically set or clear a mark. It may be more efficient to use the advanced API in this case, as it will save you from walking the tree

twice.

Advanced API

The advanced API offers more flexibility and better performance at the cost of an interface which can be harder to use and has fewer safeguards. No locking is done for you by the advanced API, and you are required to use the xa_lock while modifying the array. You can choose whether to use the xa_lock or the RCU lock while doing read-only operations on the array. You can mix advanced and normal operations on the same array; indeed the normal API is implemented in terms of the advanced API. The advanced API is only available to modules with a GPL-compatible license.

The advanced API is based around the xa_state. This is an opaque data structure which you declare on the stack using the XA_STATE() macro. This macro initialises the xa_state ready to start walking around the XArray. It is used as a cursor to maintain the position in the XArray and let you compose various operations together without having to restart from the top every time. The contents of the xa_state are protected by the rcu_read_lock() or the xas_lock(). If you need to drop whichever of those locks is protecting your state and tree, you must call xas_pause() so that future calls do not rely on the parts of the state which were left unprotected.

The xa_state is also used to store errors. You can call xas_error() to retrieve the error. All operations check whether the xa_state is in an error state before proceeding, so there's no need for you to check for an error after each call; you can make multiple calls in succession and only check at a convenient point. The only errors currently generated by the XArray code itself are `ENOMEM` and `EINVAL`, but it supports arbitrary errors in case you want to call xas_set_err() yourself.

If the xa_state is holding an `ENOMEM` error, calling xas_nomem() will attempt to allocate more memory using the specified gfp flags and cache it in the xa_state for the next attempt. The idea is that you take the xa_lock, attempt the operation and drop the lock. The operation attempts to allocate memory while holding the lock, but it is more likely to fail. Once you have dropped the lock, xas_nomem() can try harder to allocate more memory. It will return `true` if it is worth retrying the operation (i.e. that there was a memory error *and* more memory was allocated). If it has previously allocated memory, and that memory wasn't used, and there is no error (or some error that isn't `ENOMEM`), then it will free the memory previously allocated.

Internal Entries

The XArray reserves some entries for its own purposes. These are never exposed through the normal API, but when using the advanced API, it's possible to see them. Usually the best way to handle them is to pass them to xas_retry(), and retry the operation if it returns `true`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\core-api\(linux-master)(Documentation)(core-api)xarray.rst`, line 357)

Unknown directive type "flat-table".

```
.. flat-table::
   :widths: 1 1 6

   * - Name
     - Test
     - Usage

   * - Node
     - xa_is_node()
     - An XArray node.  May be visible when using a multi-index xa_state.

   * - Sibling
     - xa_is_sibling()
     - A non-canonical entry for a multi-index entry.  The value indicates
       which slot in this node has the canonical entry.

   * - Retry
     - xa_is_retry()
     - This entry is currently being modified by a thread which has the
       xa_lock.  The node containing this entry may be freed at the end
       of this RCU period.  You should restart the lookup from the head
       of the array.

   * - Zero
     - xa_is_zero()
     - Zero entries appear as ``NULL`` through the Normal API, but occupy
       an entry in the XArray which can be used to reserve the index for
       future use.  This is used by allocating XArrays for allocated entries
       which are ``NULL``.
```

Other internal entries may be added in the future. As far as possible, they will be handled by xas_retry().

Additional functionality

The xas_create_range() function allocates all the necessary memory to store every entry in a range. It will set ENOMEM in the xa_state if it cannot allocate memory.

You can use xas_init_marks() to reset the marks on an entry to their default state. This is usually all marks clear, unless the XArray is marked with `XA_FLAGS_TRACK_FREE`, in which case mark 0 is set and all other marks are clear. Replacing one entry with another using xas_store() will not reset the marks on that entry; if you want the marks reset, you should do that explicitly.

The xas_load() will walk the xa_state as close to the entry as it can. If you know the xa_state has already been walked to the entry and need to check that the entry hasn't changed, you can use xas_reload() to save a function call.

If you need to move to a different index in the XArray, call xas_set(). This resets the cursor to the top of the tree, which will generally make the next operation walk the cursor to the desired spot in the tree. If you want to move to the next or previous index, call xas_next() or xas_prev(). Setting the index does not walk the cursor around the array so does not require a lock to be held, while moving to the next or previous index does.

You can search for the next present entry using xas_find(). This is the equivalent of both xa_find() and xa_find_after(); if the cursor has been walked to an entry, then it will find the next entry after the one currently referenced. If not, it will return the entry at the index of the xa_state. Using xas_next_entry() to move to the next present entry instead of xas_find() will save a function call in the majority of cases at the expense of emitting more inline code.

The xas_find_marked() function is similar. If the xa_state has not been walked, it will return the entry at the index of the xa_state, if it is marked. Otherwise, it will return the first marked entry after the entry referenced by the xa_state. The xas_next_marked() function is the equivalent of xas_next_entry().

When iterating over a range of the XArray using xas_for_each() or xas_for_each_marked(), it may be necessary to temporarily stop the iteration. The xas_pause() function exists for this purpose. After you have done the necessary work and wish to resume, the xa_state is in an appropriate state to continue the iteration after the entry you last processed. If you have interrupts disabled while iterating, then it is good manners to pause the iteration and reenable interrupts every `XA_CHECK_SCHED` entries.

The xas_get_mark(), xas_set_mark() and xas_clear_mark() functions require the xa_state cursor to have been moved to the appropriate location in the XArray; they will do nothing if you have called xas_pause() or xas_set() immediately before.

You can call xas_set_update() to have a callback function called each time the XArray updates a node. This is used by the page cache workingset code to maintain its list of nodes which contain only shadow entries.

Multi-Index Entries

The XArray has the ability to tie multiple indices together so that operations on one index affect all indices. For example, storing into any index will change the value of the entry retrieved from any index. Setting or clearing a mark on any index will set or clear the mark on every index that is tied together. The current implementation only allows tying ranges which are aligned powers of two together; eg indices 64-127 may be tied together, but 2-6 may not be. This may save substantial quantities of memory; for example tying 512 entries together will save over 4kB.

You can create a multi-index entry by using XA_STATE_ORDER() or xas_set_order() followed by a call to xas_store(). Calling xas_load() with a multi-index xa_state will walk the xa_state to the right location in the tree, but the return value is not meaningful, potentially being an internal entry or `NULL` even when there is an entry stored within the range. Calling xas_find_conflict() will return the first entry within the range or `NULL` if there are no entries in the range. The xas_for_each_conflict() iterator will iterate over every entry which overlaps the specified range.

If xas_load() encounters a multi-index entry, the xa_index in the xa_state will not be changed. When iterating over an XArray or calling xas_find(), if the initial index is in the middle of a multi-index entry, it will not be altered. Subsequent calls or iterations will move the index to the first index in the range. Each entry will only be returned once, no matter how many indices it occupies.

Using xas_next() or xas_prev() with a multi-index xa_state is not supported. Using either of these functions on a multi-index entry will reveal sibling entries; these should be skipped over by the caller.

Storing `NULL` into any index of a multi-index entry will set the entry at every index to `NULL` and dissolve the tie. A multi-index entry can be split into entries occupying smaller ranges by calling xas_split_alloc() without the xa_lock held, followed by taking the lock and calling xas_split().

Functions and structures

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\core-api\(linux-master)(Documentation)(core-api)xarray.rst, line 495)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/xarray.h
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\core-api\(linux-master)(Documentation)(core-api)xarray.rst, line 496)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: lib/xarray.c
```