Asynchronous Transfers/Transforms API

---

1 INTRODUCTION

The async_tx API provides methods for describing a chain of asynchronous bulk memory transfers/transforms with support for inter-transactional dependencies. It is implemented as a dmaengine client that smooths over the details of different hardware offload engine implementations. Code that is written to the API can optimize for asynchronous operation and the API will fit the chain of operations to the available offload resources.

2 GENEALOGY

The API was initially designed to offload the memory copy and xor-parity-calculations of the md-raid5 driver using the offload engines present in the Intel(R) Xscale series of I/O processors. It also built on the 'dmaengine' layer developed for offloading memory copies in the network stack using Intel(R) I/OAT engines. The following design features surfaced as a result:
1/ implicit synchronous path: users of the API do not need to know if
   the platform they are running on has offload capabilities. The
   operation will be offloaded when an engine is available and carried out
   in software otherwise.
2/ cross channel dependency chains: the API allows a chain of dependent
   operations to be submitted, like xor->copy->xor in the raid5 case. The
   API automatically handles cases where the transition from one operation
   to another implies a hardware channel switch.
3/ dmaengine extensions to support multiple clients and operation types
   beyond 'memcpy'

3 USAGE

3.1 General format of the API:
```
struct dma_async_tx_descriptor *
```

async_<operation>(<op specific parameters>, struct async_submit ctl *submit)

3.2 Supported operations:

```
memcpy  - memory copy between a source and a destination buffer
memset  - fill a destination buffer with a byte value
xor     - xor a series of source buffers and write the result to a
          destination buffer
xor_val - xor a series of source buffers and set a flag if the
          result is zero.  The implementation attempts to prevent
          writes to memory
pq      - generate the p+q (raid6 syndrome) from a series of source buffers
pq_val  - validate that a p and or q buffer are in sync with a given series of
          sources
datap   - (raid6_datap_recov) recover a raid6 data block and the p block
          from the given sources
2data   - (raid6_2data_recov) recover 2 raid6 data blocks from the given
          sources
```

3.3 Descriptor management:
The return value is non-NULL and points to a 'descriptor' when the operation has been queued to execute asynchronously. Descriptors are recycled resources, under control of the offload engine driver, to be reused as operations complete. When an application needs to submit a chain of operations it must guarantee that the descriptor is not automatically recycled before the dependency is submitted. This requires that all descriptors be acknowledged by the application before the offload engine driver is allowed to recycle (or free) the descriptor. A descriptor can be acked by one of the following methods:
1/ setting the ASYNC_TX_ACK flag if no child operations are to be submitted
2/ submitting an unacknowledged descriptor as a dependency to another async_tx call will implicitly set the acknowledged state.
3/ calling async_tx_ack() on the descriptor.

3.4 When does the operation execute?
Operations do not immediately issue after return from the async_<operation> call. Offload engine drivers batch operations to improve performance by reducing the number of mmio cycles needed to manage the channel. Once a driver-specific threshold is met the driver automatically issues pending operations. An application can force this event by calling async_tx_issue_pending_all(). This operates on all channels since the application has no knowledge of channel to operation mapping.

3.5 When does the operation complete?
There are two methods for an application to learn about the completion of an operation.
1/ Call dma_wait_for_async_tx(). This call causes the CPU to spin while it polls for the completion of the operation. It handles dependency chains and issuing pending operations.
2/ Specify a completion callback. The callback routine runs in tasklet context if the offload engine driver supports interrupts, or it is called in application context if the operation is carried out synchronously in software. The callback can be set in the call to async_<operation>, or when the application needs to submit a chain of unknown length it can use the async_trigger_callback() routine to set a completion interrupt/callback at the end of the chain.

3.6 Constraints:
1/ Calls to async_<operation> are not permitted in IRQ context. Other
   contexts are permitted provided constraint #2 is not violated.
2/ Completion callback routines cannot submit new operations. This
   results in recursion in the synchronous case and spin_locks being
   acquired twice in the asynchronous case.

3.7 Example:
Perform a xor->copy->xor operation where each operation depends on the
result from the previous operation:

```
void callback(void *param)
{
        struct completion *cmp = param;

        complete(cmp);
}

void run_xor_copy_xor(struct page **xor_srcs,
                      int xor_src_cnt,
                      struct page *xor_dest,
                      size_t xor_len,
                      struct page *copy_src,
                      struct page *copy_dest,
                      size_t copy_len)
{
        struct dma_async_tx_descriptor *tx;
        addr_conv_t addr_conv[xor_src_cnt];
        struct async_submit_ctl submit;
        addr_conv_t addr_conv[NDISKS];
        struct completion cmp;

        init_async_submit(&submit, ASYNC_TX_XOR_DROP_DST, NULL, NULL, NULL,
                          addr_conv);
        tx = async_xor(xor_dest, xor_srcs, 0, xor_src_cnt, xor_len, &submit)

        submit->depend_tx = tx;
        tx = async_memcpy(copy_dest, copy_src, 0, 0, copy_len, &submit);

        init_completion(&cmp);
        init_async_submit(&submit, ASYNC_TX_XOR_DROP_DST | ASYNC_TX_ACK, tx,
                          callback, &cmp, addr_conv);
        tx = async_xor(xor_dest, xor_srcs, 0, xor_src_cnt, xor_len, &submit);

        async_tx_issue_pending_all();

        wait_for_completion(&cmp);
}
```

See include/linux/async_tx.h for more information on the flags. See the
ops_run_* and ops_complete_* routines in drivers/md/raid5.c for more
implementation examples.

4 DRIVER DEVELOPMENT NOTES

4.1 Conformance points:
There are a few conformance points required in dmaengine drivers to accommodate assumptions made by applications using the async_tx API:
1/ Completion callbacks are expected to happen in tasklet context
2/ dma_async_tx_descriptor fields are never manipulated in IRQ context
3/ Use async_tx_run_dependencies() in the descriptor clean up path to
   handle submission of dependent operations

4.2 "My application needs exclusive control of hardware channels"
Primarily this requirement arises from cases where a DMA engine driver is being used to support device-to-memory operations. A channel that is performing these operations cannot, for many platform specific reasons, be shared. For these cases the dma_request_channel() interface is provided.

The interface is:
```
struct dma_chan *dma_request_channel(dma_cap_mask_t mask,
                                     dma_filter_fn filter_fn,
                                     void *filter_param);
```

Where dma_filter_fn is defined as:
```
typedef bool (*dma_filter_fn)(struct dma_chan *chan, void *filter_param);
```

When the optional 'filter_fn' parameter is set to NULL dma_request_channel simply returns the first channel that satisfies the capability mask. Otherwise, when the mask parameter is insufficient for specifying the necessary channel, the filter_fn routine can be used to disposition the available channels in the system. The filter_fn routine is called once for each free channel in the system. Upon seeing a suitable channel filter_fn returns DMA_ACK which flags that channel to be the return value from dma_request_channel. A channel allocated via this interface is exclusive to the caller, until dma_release_channel() is called.

The DMA_PRIVATE capability flag is used to tag dma devices that should not be used by the general-purpose allocator. It can be set at initialization time if it is known that a channel will always be private. Alternatively, it is set when dma_request_channel() finds an unused "public" channel.

A couple caveats to note when implementing a driver and consumer:
1/ Once a channel has been privately allocated it will no longer be
   considered by the general-purpose allocator even after a call to
   dma_release_channel().
2/ Since capabilities are specified at the device level a dma_device
   with multiple channels will either have all channels public, or all
   channels private.

5 SOURCE

include/linux/dmaengine.h: core header file for DMA drivers and api users
drivers/dma/dmaengine.c: offload engine channel management routines
drivers/dma/: location for offload engine drivers
include/linux/async_tx.h: core header file for the async_tx api
crypto/async_tx/async_tx.c: async_tx interface to dmaengine and common code
crypto/async_tx/async_memcpy.c: copy offload

crypto/async_tx/async_memset.c: memory fill offload
crypto/async_tx/async_xor.c: xor and xor zero sum offload