MMC Asynchronous Request

Rationale

How significant is the cache maintenance overhead?

It depends. Fast eMMC and multiple cache levels with speculative cache pre-fetch makes the cache overhead relatively significant. If the DMA preparations for the next request are done in parallel with the current transfer, the DMA preparation overhead would not affect the MMC performance.

The intention of non-blocking (asynchronous) MMC requests is to minimize the time between when an MMC request ends and another MMC request begins.

Using mmc_wait_for_req(), the MMC controller is idle while dma_map_sg and dma_unmap_sg are processing. Using non-blocking MMC requests makes it possible to prepare the caches for next job in parallel with an active MMC request.

MMC block driver

The mmc_blk_issue_rw_rq() in the MMC block driver is made non-blocking.

The increase in throughput is proportional to the time it takes to prepare (major part of preparations are dma_map_sg() and dma_unmap_sg()) a request and how fast the memory is. The faster the MMC/SD is the more significant the prepare request time becomes. Roughly the expected performance gain is 5% for large writes and 10% on large reads on a L2 cache platform. In power save mode, when clocks run on a lower frequency, the DMA preparation may cost even more. As long as these slower preparations are run in parallel with the transfer performance won't be affected.

Details on measurements from IOZone and mmc_test

https://wiki.linaro.org/WorkingGroups/Kernel/Specs/StoragePerfMMC-async-req

MMC core API extension

There is one new public function mmc_start_req().

It starts a new MMC command request for a host. The function isn't truly non-blocking. If there is an ongoing async request it waits for completion of that request and starts the new one and returns. It doesn't wait for the new request to complete. If there is no ongoing request it starts the new request and returns immediately.

MMC host extensions

There are two optional members in the mmc_host_ops -- pre_req() and post_req() -- that the host driver may implement in order to move work to before and after the actual mmc_host_ops.request() function is called.

In the DMA case pre_req() may do dma_map_sg() and prepare the DMA descriptor, and post_req() runs the dma_unmap_sg().

Optimize for the first request

The first request in a series of requests can't be prepared in parallel with the previous transfer, since there is no previous request.

The argument is_first_req in pre_req() indicates that there is no previous request. The host driver may optimize for this scenario to minimize the performance loss. A way to optimize for this is to split the current request in two chunks, prepare the first chunk and start the request, and finally prepare the second chunk and start the transfer.

Pseudocode to handle is_first_req scenario with minimal prepare overhead:

```
if (is_first_req && req->size > threshold)
   /* start MMC transfer for the complete transfer size */
   mmc_start_command(MMC_CMD_TRANSFER_FULL_SIZE);

   /*
    * Begin to prepare DMA while cmd is being processed by MMC.
    * The first chunk of the request should take the same time
    * to prepare as the "MMC process command time".
    * If prepare time exceeds MMC cmd time
    * the transfer is delayed, guesstimate max 4k as first chunk size.
    */
    prepare_1st_chunk_for_dma(req);
    /* flush pending desc to the DMAC (dmaengine.h) */
    dma_issue_pending(req->dma_desc);

    prepare_2nd_chunk_for_dma(req);
```

```
/*
 * The second issue_pending should be called before MMC runs out
 * of the first chunk. If the MMC runs out of the first data chunk
 * before this call, the transfer is delayed.
 */
dma_issue_pending(req->dma_desc);
```