```c
static unsigned ifs_find_dirty_range(struct folio *folio,
                struct iomap_folio_state *ifs, u64 *range_start, u64 range_end)
{
        struct inode *inode = folio->mapping->host;
        unsigned start_blk =
                offset_in_folio(folio, *range_start) >> inode->i_blkbits;
        unsigned end_blk = min_not_zero(
                offset_in_folio(folio, range_end) >> inode->i_blkbits,
                i_blocks_per_folio(inode, folio));
        unsigned nblks = 1;

        while (!ifs_block_is_dirty(folio, ifs, start_blk))
                if (++start_blk == end_blk)
                        return 0;

        while (start_blk + nblks < end_blk) {
                if (!ifs_block_is_dirty(folio, ifs, start_blk + nblks))
                        break;
                nblks++;
        }

        *range_start = folio_pos(folio) + (start_blk << inode->i_blkbits);
        return nblks << inode->i_blkbits;
}
static unsigned iomap_find_dirty_range(struct folio *folio, u64 *range_start,
                u64 range_end)
{
        struct iomap_folio_state *ifs = folio->private;

        if (*range_start >= range_end)
                return 0;

        if (ifs)
                return ifs_find_dirty_range(folio, ifs, range_start, range_end);
        return range_end - *range_start;
}
unsigned f2fs_iomap_find_dirty_range(struct folio *folio, u64 *range_start,
                u64 range_end)
{
        struct inode* inode=folio->mapping->host;
        if(f2fs_compressed_file(inode))
        {
                range_end=min(range_end,(range_end>>F2FS_I(inode)->i_log_cluster_size+1)<< F2FS_I(
        }
        return iomap_find_dirty_range(folio, range_start, range_end);
}
```

```
static int f2fs_write_single_folio(struct folio *folio, int *submitted_pages_count,
                                struct bio **bio_ptr, sector_t *last_block_ptr,
                                struct writeback_control *wbc,
                                enum iostat_type io_type,
                                unsigned int compr_blocks, bool is_reclaim,
                                u64 dirty_pos_start, u64 dirty_pos_end)
{
        struct inode *inode = folio->mapping->host;
        struct f2fs_sb_info *sbi = F2FS_I_SB(inode);
        struct f2fs_iomap_folio_state *ifs = folio->private; // ifs should be set by caller if fol
        pgoff_t folio_start_index = folio->index;
        int err = 0;
        int local_submitted_count = 0;

        // Calculate the page indices within the folio that correspond to the dirty range
        // The dirty_pos_start/end are file offsets. We need folio-relative page indices.
        pgoff_t first_page_idx_in_folio = (dirty_pos_start - folio_pos(folio)) >> PAGE_SHIFT;
        pgoff_t last_page_idx_in_folio = (dirty_pos_end - 1 - folio_pos(folio)) >> PAGE_SHIFT;

        for (pgoff_t i = first_page_idx_in_folio; i <= last_page_idx_in_folio; ++i) {
                struct page *page = folio_page(folio, i);
                pgoff_t current_page_file_index = folio_start_index + i;

                // Skip if page is beyond EOF (i_size_read is expensive here,
                // writeback_iter and iomap_writepage_handle_eof should have handled major EOF iss
                // However, a check against wbc->range_end or a precise end_index might be useful
                // if not fully covered by iomap_find_dirty_range logic)
                if (current_page_file_index >= wbc_folio_end_index(wbc, folio)) {
                        // This case should ideally be prevented by iomap_find_dirty_range
                        // or iomap_writepage_handle_eof. If it occurs, it means the
                        // dirty range extends beyond what wbc expects.
                        continue;
                }

                // For large folios, increment pending bytes before attempting to write the page
                if (ifs) {
                        atomic_add(PAGE_SIZE, &ifs->write_bytes_pending);
                }
                struct f2fs_io_info fio = {
                        .sbi = sbi,
                        .ino = inode->i_ino,
                        .type = DATA, // Assuming DATA type, adjust if node pages are handled here
                        .op = REQ_OP_WRITE,
                        .op_flags = wbc_to_write_flags(wbc),
                        .old_blkaddr = NULL_ADDR,
                        .page = page,
                        .encrypted_page = NULL,
                        .submitted = 0, // f2fs_do_write_data_page will update this if it submits
                        .compr_blocks = compr_blocks, // Pass this through
                        .need_lock = compr_blocks ? LOCK_DONE : LOCK_RETRY,
                        .meta_gc = f2fs_meta_inode_gc_required(inode) ? 1 : 0,
                        .io_type = io_type,
                        .io_wbc = wbc,
                        // For bio and last_block, F2FS has its own merging.
                        // If f2fs_do_write_data_page uses them directly, pass pointers.
                        // If it's about merging *across calls* to f2fs_do_write_data_page,
                        // that logic is more complex and currently sits higher up or in IPU.
                        // For simplicity here, let's assume f2fs_do_write_data_page
                        // handles its bio needs internally or via fio->bio.
                        // The bio_ptr and last_block_ptr are from the original f2fs_write_single_
                        // which might have tried to merge. Here, each page is distinct for fio.
                        .bio = bio_ptr ? *bio_ptr : NULL, // Allow passing bio for potential inter
                        .last_block = last_block_ptr ? *last_block_ptr : 0,
                };

                // Retain original retry logic for f2fs_do_write_data_page
                err = f2fs_do_write_data_page(&fio);
                if (err == -EAGAIN) {
                        f2fs_bug_on(sbi, compr_blocks && err == -EAGAIN); // Original bug_on
                        fio.need_lock = LOCK_REQ;
                        err = f2fs_do_write_data_page(&fio);
                }

                if (bio_ptr) // Update the caller's bio if f2fs_do_write_data_page modified it
                        *bio_ptr = fio.bio;
                if (last_block_ptr)
                        *last_block_ptr = fio.last_block;
                if (err) {
                        // If this page failed to be prepared for write (e.g. -ENOSPC),
                        // decrement pending_bytes if we incremented it.
                        if (ifs) {
                                // If submission failed for this page, it won't go to f2fs_write_e
                                // for this portion.
                                if (atomic_sub_and_test(PAGE_SIZE, &ifs->write_bytes_pending)) {
                                        // This implies all other pending IO for this folio also c
                                        // and this was the last one.
                                        // This scenario is tricky: if other pages *were* submitte
                                        // they will call folio_end_writeback. If this is the *onl
                                        // page and it fails, then folio_end_writeback needs to be
                                        // by the caller (f2fs_write_cache_folios) after unlock.
                                        // The iomap model handles this by a bias count, which is
                                        // decremented after the loop. If it hits 0, and no IO was
                                        // then it calls folio_end_writeback.
                                        // For now, just decrement. The caller will handle overall
                                }
                        }
                        // An error on one page within the folio typically means we stop
                        // processing this folio and report the error.
                        // The VFS writeback loop (writeback_iter) might then requeue the folio.
                        mapping_set_error(folio->mapping, err); // Report error to mapping
                        break; // Stop processing this folio on error
                } else {
                        // Page successfully processed by f2fs_do_write_data_page
                        // fio.submitted should reflect if f2fs_do_write_data_page actually
                        // queued something (it's usually 1 for a single page, or more if it did i
                        // The original f2fs_write_single_data_page updated *submitted_count.
                        // Here, we count pages for which f2fs_do_write_data_page succeeded.
                        local_submitted_count++;
                        if (wbc->nr_to_write > 0) // Only decrement if wbc cares
                                wbc->nr_to_write--; // Decrement for each page successfully handle
                }

                // If wbc->nr_to_write is a budget and we've exhausted it.
                if (wbc->nr_to_write == 0 && wbc->sync_mode == WB_SYNC_NONE) {
```

~~~
                        err = -EAGAIN; // Signal to stop, budget met
                        break;
                }
        }

        if (submitted_pages_count)
                *submitted_pages_count = local_submitted_count;

        return err;
}
```C

```C
static int f2fs_write_cache_folios(struct address_space *mapping,
                                   struct writeback_control *wbc,
                                   enum iostat_type io_type)
{
        struct folio *folio =
                NULL;
        int err = 0;
        struct inode *inode = mapping->host;
        bool is_compressed_file = f2fs_compressed_file(inode);
        struct compress_ctx cc;
        int iter_err = 0; // Error from writeback_iter
#ifdef CONFIG_F2FS_FS_COMPRESSION
        if (is_compressed_file) {
                // Initialize compress_ctx structure fields (inode, sizes, etc.)
                cc.inode = inode;
                cc.log_cluster_size = F2FS_I(inode)->i_log_cluster_size;
                cc.cluster_size = 1 << cc.log_cluster_size;
                // cc.rpages and cc.cpages will be allocated by f2fs_init_compress_ctx
                cc.rpages = NULL;
                cc.cpages = NULL;
                cc.cluster_idx = NULL_CLUSTER;
                cc.nr_rpages = 0;
                cc.nr_cpages = 0;
                cc.valid_nr_cpages = 0;
                if (f2fs_init_compress_ctx(&cc)) {
                        is_compressed_file = false;
~~~

```
                }
        }
#endif
        if (get_dirty_pages(mapping->host) <=
            SM_I(F2FS_M_SB(mapping))->min_hot_blocks)
                set_inode_flag(mapping->host, FI_HOT_DATA);
        else
                clear_inode_flag(mapping->host, FI_HOT_DATA);

        while ((folio = writeback_iter(mapping, wbc, folio, &iter_err))) {
                struct f2fs_iomap_folio_state *fifs = NULL;
                u64 pos = folio_pos(folio);
                u64 end_pos = pos + folio_size(folio);
                u64 end_pos_aligned;
                u32 r_len;
                int submitted_pages_this_folio = 0;
                if (!iomap_writepage_handle_eof(folio, inode, &end_pos)) {
                        folio_unlock(folio);
                        return 0;
                }

                fifs = folio->private;
                if (i_blocks_per_folio(inode, folio) > 1) {
                        if (!fifs) {
                                fifs = fifs_alloc(inode, folio, 0);
                                iomap_set_range_dirty(folio, 0, end_pos - pos);
                        }

                        /*
                 * Keep the I/O completion handler from clearing the writeback
                 * bit until we have submitted all blocks by adding a bias to
                 * fifs->write_bytes_pending, which is dropped after submitting
                 * all blocks.
                 */
                        WARN_ON_ONCE(atomic_read(&fifs->write_bytes_pending) !=
                                     0);
                        atomic_inc(&fifs->write_bytes_pending);
                }
                folio_start_writeback(folio);
                end_aligned = round_up(end_pos, i_blocksize(inode));
                while ((r_len = f2fs_iomap_find_dirty_range(folio, &pos,
                                                       end_aligned))) {
                        if (is_compressed_file)
                        {
                                pgoff_t cluster_idx =
                                        cluster_idx(&cc, pos >> PAGE_SHIFT);

                                // If current dirty range belongs to a new cluster than what's in
                                if (cc.cluster_idx != NULL_CLUSTER &&
                                    cc.cluster_idx != cluster_idx) {
                                        int submitted_compress = 0;
                                        if (!f2fs_cluster_is_empty(&cc)) {
                                                err = f2fs_write_multi_pages(
                                                        &cc,
                                                        &submitted_compress,
                                                        wbc, io_type);
                                                // wbc->nr_to_write -= submitted_compress;
                                                /* writeback_iter choose to dec folio_nr_pages in
                                                 * completly different from the dec policy that f2
                                                 * it's an open problem here
                                                */
                                        }
                                        f2fs_destroy_compress_ctx(
                                                &cc,
                                                false); // false to reuse buffers, resets nr_rpage
                                        f2fs_init_compress_ctx(
                                                &cc); // Ensure rpages is valid
                                        cc.cluster_idx =
                                                NULL_CLUSTER; // Explicitly ensure for next logic
                                }
                                /*我们暂时没完全搞清prepare_compress_overwrite
                                的防御性编程在f2fs_write_cache_folios中如何实现等价逻辑。先注释掉吧*/
                                // // If cc is now for a new/empty cluster, prepare it
                                // if (f2fs_cluster_is_empty(&cc)) {
                                //         cc.cluster_idx =
                                //                 cluster_idx; // Set for prepare_compress_overwr
                                //         int prep_ret = prepare_compress_overwrite(
                                //                 &cc, NULL,
                                //                 pos >>
                                //                         PAGE_SHIFT, /* pgoff_t index */
                                //                 NULL);
                                //         if (prep_ret < 0) {
                                //                 err = prep_ret;
                                //                 goto handle_current_folio_error;
                                //         }
                                //         // After prepare_compress_overwrite, cc.cluster_idx is
                                //         // and cc.rpages might be filled if it was an overwrite
                                // }

                                // Add the current dirty range from folio to cc
                                // f2fs_compress_ctx_add_folio does not do folio_get.
                                // The folio is locked. cc needs a ref.
                                folio_get(folio); // cc takes a reference
                                f2fs_compress_ctx_add_folio(&cc, folio,pos,r_len);
                                // The ref taken by folio_get will be put by f2fs_put_rpages_wbc (
                                /*靠判断相邻两次cluster不连续 然后就进行f2fs_write_multi_pages的
                                那个逻辑,是足够覆盖掉cluster满的这个逻辑的*/
                                // if (f2fs_cluster_is_full(&cc)) {
                                //         int submitted_compress = 0;
                                //         err = f2fs_write_multi_pages(
                                //                 &cc, &submitted_compress, wbc,
                                //                 io_type);
                                //         // wbc->nr_to_write -= submitted_compress;
                                //         if (err) {
                                //                 if (err == -EAGAIN)
                                //                         wbc->nr_to_write = 0;
                                //                 goto handle_current_folio_error;
                                //         }
                                //         f2fs_destroy_compress_ctx(
                                //                 &cc,
                                //                 false); // Reset for next potential cluster
                                //         f2fs_init_compress_ctx(&cc);
                                //         cc.cluster_idx = NULL_CLUSTER;
                                // }
                        }
                        else
```

```
                        { // Non-compressed file path
                                int submitted_this_range = 0;
                                // For non-compressed, f2fs_write_single_folio handles one dirty r
                                // It internally creates f2fs_io_info (fio) and calls f2fs_do_writ
                                // It needs the folio (folio), and the dirty range.
                                err = f2fs_write_single_folio(folio, &submitted_this_range, NULL,N
                                // submitted_pages_this_folio += submitted_this_range; // f2fs_wri
                        }
                        pos += r_len; // Advance for next iomap_find_dirty_range call
                } // End of while(iomap_find_dirty_range)

                // After processing all dirty ranges for this folio
                iomap_clear_range_dirty(folio, 0, folio_size(folio));

handle_current_folio_error: // Common error handling for the current folio
                folio_unlock(folio);
                if (fifs) {
                        if (atomic_dec_and_test(&fifs->write_bytes_pending)) {
                                folio_end_writeback(folio);
                        }
                } else {
                        if (!err &&
                            pos ==
                                    current_folio_pos_in_file) { // No dirty ranges processed
                                folio_end_writeback(folio);
                        } else if (err && err != -EAGAIN) {
                                // If an error stopped processing before any IO for this folio,
                                // or if f2fs_write_single_folio failed before submitting.
                                // This needs to be robust.
                                // For now, rely on fifs or assume single-page folios' end_io hand
                        }
                }
                if (err &&
                    err != -EAGAIN) { // If a real error occurred for this folio
                        mapping_set_error(mapping, err);
                        // writeback_iter will stop if wbc->sync_mode != WB_SYNC_NONE on error.
                        // Or we can break here.
                        break;
                }
```

```
        } // End of while(writeback_iter)

        if (is_compressed_file && !f2fs_cluster_is_empty(&cc)) {
                int submitted_compress = 0;
                int flush_err = f2fs_write_multi_pages(&cc, &submitted_compress,
                                                       wbc, io_type);
                // wbc->nr_to_write -= submitted_compress;
                if (flush_err && !err) { // Prioritize earlier errors
                        err = flush_err;
                }
        }

        if (is_compressed_file) {
                // Free cc.rpages and cc.cpages if they were allocated
                f2fs_destroy_compress_ctx(
                        &cc,
                        true); // true to free cpages, rpages also freed by page_array_free
        }

        // If err was -EAGAIN from writeback_iter (iter_err) or our loop, VFS expects 0.
        if (err == -EAGAIN || iter_err == -EAGAIN) {
                return 0;
        }
        return err ? err : iter_err;
}
```

```
static int prepare_compress_overwrite(struct compress_ctx *cc,
                struct page **pagep, pgoff_t index, void **fsdata)
{
        struct f2fs_sb_info *sbi = F2FS_I_SB(cc->inode);
        struct address_space *mapping = cc->inode->i_mapping;
        struct folio *folio;
        sector_t last_block_in_bio;
        fgf_t fgp_flag = FGP_LOCK | FGP_WRITE | FGP_CREAT;
        pgoff_t start_idx = start_idx_of_cluster(cc);
        int i, ret;

retry:
        ret = f2fs_is_compressed_cluster(cc->inode, start_idx);
        if (ret <= 0)
                return ret;

        ret = f2fs_init_compress_ctx(cc);
        if (ret)
                return ret;
        /* keep folio reference to avoid page reclaim */
        for (i = 0; i < cc->cluster_size; ) {
                folio = f2fs_filemap_get_folio(mapping, start_idx + i,
                                fgp_flag, GFP_NOFS);
                if (IS_ERR(folio)) {
                        ret = PTR_ERR(folio);
                        goto unlock_pages;
                }
                bool needs_read = false;
                pgoff_t idx_in_folio=offset_in_folio(folio,(start_idx+i)<<PAGE_SHIFT)>>PAGE_SHIFT;
                unsigned int num_pages_iter=min(folio_nr_pages(folio)-idx_in_folio,cc->cluster_siz
                for (; idx_in_folio < num_pages_iter; ++idx_in_folio) {

                        struct f2fs_iomap_folio_state* fifs=folio->private;
                        if (!f2fs_ifs_block_is_uptodate(fifs, idx_in_folio)) {
                                needs_read = true;
                                break;
                        }

                }
                if(needs_read)
                {
                        f2fs_compress_ctx_add_folio(cc,folio,(folio->index + idx_in_folio) << PAGE
                }
                else
                {
                        folio_put(folio);
                }
                i+=num_pages_iter;
        }

        if (!f2fs_cluster_is_empty(cc)) {
                struct bio *bio = NULL;

                ret = f2fs_read_multi_folios(cc, &bio, cc->cluster_size,
                                        &last_block_in_bio, NULL, true);
                f2fs_put_rpages(cc);
                f2fs_destroy_compress_ctx(cc, true);
                if (ret)
                        goto out;
                if (bio)
                        f2fs_submit_read_bio(sbi, bio, DATA);

                ret = f2fs_init_compress_ctx(cc);
                if (ret)
                        goto out;
        }

        for (i = 0; i < cc->cluster_size; ) {
                f2fs_bug_on(sbi, cc->rpages[i]);

                folio = filemap_lock_folio(mapping, start_idx + i);
                if (IS_ERR(folio)) {
                        /* folio could be truncated */
                        goto release_and_retry;
                }

                f2fs_folio_wait_writeback(folio, DATA, true, true);
                pgoff_t idx_in_folio=offset_in_folio(folio,(start_idx+i)<<PAGE_SHIFT)>>PAGE_SHIFT;
                unsigned int num_pages_to_add=min(folio_nr_pages(folio)-idx_in_folio,cc->cluster_s
                f2fs_compress_ctx_add_folio(cc, folio,(start_idx + i) >> PAGE_SHIFT,num_pages_to_a

                if (!folio_test_uptodate(folio)) {
                        f2fs_handle_page_eio(sbi, folio, DATA);
release_and_retry:
                        f2fs_put_rpages(cc);
                        f2fs_unlock_rpages(cc, i + 1);
                        f2fs_destroy_compress_ctx(cc, true);
                        goto retry;
                }
                i+=num_pages_to_add;
        }

        if (likely(!ret)) {
                *fsdata = cc->rpages;
                *pagep = cc->rpages[offset_in_cluster(cc, index)];
                return cc->cluster_size;
        }

unlock_pages:
        f2fs_put_rpages(cc);
        f2fs_unlock_rpages(cc, i);
        f2fs_destroy_compress_ctx(cc, true);
out:
        return ret;
}
```

```
static void f2fs_drop_rpages(struct compress_ctx *cc, int len, bool unlock)
{
        int i;
        unsigned int num_to_skip=0;
        for (i = 0; i < len; ) {
                if (!cc->rpages[i])
                {
                        num_to_skip = 1;
                        continue;
                }
                struct folio* rfolio=page_folio(cc->rpages[i]);
                if (unlock)
                        folio_unlock(rfolio);
                else
                        folio_put(rfolio);
                num_to_skip =min(folio_nr_pages(rfolio)-folio_page_idx(rfolio,cc->rpages[i]), len-
                i+=num_to_skip;
        }
}

static void f2fs_put_rpages(struct compress_ctx *cc)
{
        f2fs_drop_rpages(cc, cc->cluster_size, false);
}

static void f2fs_unlock_rpages(struct compress_ctx *cc, int len)
{
        f2fs_drop_rpages(cc, len, true);
}

static void f2fs_put_rpages_wbc(struct compress_ctx *cc,
                struct writeback_control *wbc, bool redirty, int unlock)
{
        int i;
        unsigned int num_to_skip=0;
        for (i = 0; i < len; ) {
                if (!cc->rpages[i])
                {
                        num_to_skip = 1;
                        continue;
                }
                struct folio* rfolio=page_folio(cc->rpages[i]);
                if (redirty)
                        folio_redirty_for_write_page(rfolio);
                f2fs_folio_put(page_folio(cc->rpages[i]), unlock);
                num_to_skip =min(folio_nr_pages(rfolio)-folio_page_idx(rfolio,cc->rpages[i]), len-
                i+=num_to_skip;
        }
}
```