Scheduler Module

2018年8月23日 18:38

Schedule部分

整体流程

Structure

SIMQueue

SIMQueue	Size	Latency	Name	dataType	IsPtr	Description
q_set_arf_bit[threadnum]	50	setting_schedule_to_set_arf_bit		Int (rob index)	False	设置schedule到设置arf bit的时延(arf用于减小\RAT)
q_uaq	pb文件中对应的uaq entry描述的大小	pb文件中对应的uaq entry描述的时延	"name"	Int*	False	
q_rs_unknown_net_check[threadnum]	50	setting_sched_to_unknown_netcheck_latency		Int (rob index)	False	RS dispatch到unknown STA net check的时延
q_delay_sta_wakeup_load	setting_delay_sta_wakeup_of_loads	setting_delay_sta_wakeup_of_loads		Int (rob index)	False	设置依赖sta的load的wakup时延
q_delay_std_wakeup_load	setting_delay_std_wakeup_of_loads	setting_delay_std_wakeup_of_loads		Int (rob index)	False	设置依赖std的load的wakup时延
q_reuse_sched_part[sched_portnum][scheduler_num]	setting_bb_xxx_sched_part_size_list & 0xff scheduler类型 SCHEDULER_MEMORY SCHEDULER_LOAD SCHEDULER_STORE SCHEDULER_GENERAL SCHEDULER_FP SCHEDULER_FPLOAD SCHEDULER_MAX(哨兵) 目前，使用的有两类scheduler SCHEDULER_MEMORY SCHEDULER_GENERAL	setting_bb_xxx_sched_part_reuse_list & 0xff		Int *	False	
q_replay_rid[sched_portnum][scheduler_num]	replay_loop_entries	setting_replay_latency - setting_dispatch_latency		Int*	False	
q_replay_phytid	replay_loop_entries	setting_replay_latency - setting_dispatch_latency		Int*	Fasle	
q_replay_safe_rid[sched_portnum][scheduler_num]						
q_replay_safe_phytid						
q_rs_dispatch_rid	Max_uops_in_flight	Setting_dispatch_latency		Int*	False	
q_rs_dispatch_phytid						
q_mob_dispatch_rid						
q_mob_dispatch_phytid						
q_fb_bypass_ul2_wakeup_rid	(Fp_bypass_queue_latency/setting_sched_clock)+ 1	Fp_bypass_queue_latency		Int*	False	
q_fb_bypass_ul2_wakeup_phytid						

Structure

Priv

Name	Size	Description
aalu_sched_used	int	
agu_sched_used	int	

br_sched_used	int	
cnt	uint32 [SCHEDULER_MAX]	
fld_sched_used	int	
ialu_sched_used	int	
uaq_moved	bool [UAQ_MAX]	
istd_sched_used	int	
last_ld_miss_cycle	cycle	
last_sched_uop_tid	int	
uops_scheduled	int*	
mrn_sched_used	int	
mstd_sched_used	int	
num_int_dispatch_this_cycle	int	
q_reuse_sched_part	SIMQ *[MAX_SCHED_PORTS] [SCHEDULER_MAX]	
sched_pb_buffer	int **	
sta_sched_used	int	
hurricane_start	int [MAX_THREADS]	
max_wait_in_ssu	uint32 [MAX_THREADS]	
max_wait_in_ssu_fp	uint32 [MAX_THREADS]	
max_wait_in_ssu_int	uint32 [MAX_THREADS]	
max_wait_in_ssu_mem	uint32 [MAX_THREADS]	
occupied_exec_port	int **	
q_set_arf_bit	SIMQ * [MAX_THREADS]	
sched_part_active_entries	int ***	
sched_part_fat_active_entries	int **	
sched_part_max_entries	int **	
sched_part_reuse	int **	
sched_port_replayed	bool **	
sched_port_used	int **	
sched_retry_delay	int *	
sched_retry_delay_cntr	int *	
scheduler_assoc	uint32 *	
scheduler_max_uops	uint32 *	
scheduler_min_entries_for_other_threads	uint32	
scheduler_ptr	uint32 [scheduler_num] [phythread_num][MAX_ROBSIZE]	scheduler的结构数据，根据不同的settng_schedule_policy(scheduler在phythread的选择策略)，scheduler的结构不同 POLICY_THREAD_UNAWARE [scheduler_num][1][MAX_ROBSIZE] Others [scheduler_num][phythread_num] [MAX_ROBSIZE]
scheduler_rate	uint32 *	
scheduler_size	uint32 *	
scheduler_active_phytid	uint32 [scheduler_num]	和scheduler_tid_ptr是互斥关系，目前没有使用，目前policy为POLICY_THTREAD_UNAWARE
scheduler_tid_ptr	uint32 [scheduler_num] [MAX_ROBSIZE]	某个uop分配进入scheduler后，其对应的phythread id POLICY_THREAD_UNAWARE [scheduler_num][MAX_ROBSIZE] Others NULL
scheduler_unfused_active_entries[scheduler_num][num_phythread]	uint32**	表明uop已经进入scheduler，可以进行dispatch；这里的uop指的是fusing后的uop或是没有fusing的uop 这个counter在setting_new_rs_fusing的option下有两种含义 Enable setting_new_rs_fusing 只有当fusing uop的全部uop全部进入scheduler，这个counter更新 Disable setting_new_rs_fusing 当fusing的first进入scheduler的时候，这个counter更新

scheduler_unfused_total_active_entries[scheudler_num]	uint32 **	表明每个scheduler上面当前被占据的entry数量，这个不考虑被fused uop(这里应该指的是macro-fuse，不是micro-fuse)，这个不区分不同的phythread
unified_scheduler_unfused_active_entries[num_phythread]	uint32 *	这个和上面一样，不区分scheduler type
scheduler_active_entries[scheduler_num][phythread_num]	uint32 **	和上面的不同，这里仅仅考虑一个单独的uop，不考虑fusing的情况
scheduler_total_active_entries[scheudler_num]	uint32 *	
unified_scheduler_active_entries[num_phythread]	uint32 *	
thread_hurricaneed	bool [MAX_THREADS]	
thread_hurricaneed_fp	bool [MAX_THREADS]	
thread_hurricaneed_int	bool [MAX_THREADS]	
thread_hurricaneed_mem	bool [MAX_THREADS]	
hurricane_block_fp_uaq_until	uint32 [MAX_THREADS]	
hurricane_block_uaq_until	uint32 [MAX_THREADS]	
hurricane_block_int_uaq_until	uint32 [MAX_THREADS]	
hurricane_block_mem_uaq_until	uint32 [MAX_THREADS]	
hurricane_off_trigger	int	
hurricane_off_trigger_fp	int	
hurricane_off_trigger_int	int	
hurricane_off_trigger_mem	int	
hurricane_on_trigger	int	
Hurricane_on_trigger_fp	int	
hurricane_on_trigger_int	int	
hurricane_on_trigger_mem	int	
last_sched_uop_plr	Pipeline_RecordId	
scheduler_write_port_used[scheduler_num]	int *	uop分配进入当前的scheduler，所以消耗了当前scheduler的write_port
q_delay_sta_wakeup_load	SIMQ *	
q_delay_std_wakeup_load	SIMQ *	
lock_scheduled	bool	
dispatch_port_busy[sched_portnum][scheuduler_num]	bool **	表明当前T某个port是否被schedule的uop(视配置而定setting_no_sched_port_limitation(false))，每个T自动清0
lost_opportunity_score	bool **	
rf_access_per_cycle	int	
num_alu_this_cycle	uint32	
num_fp_this_cycle	uint32	
num_br_this_cycle	uint32	
num_ld_this_cycle	uint32	
num_sta_this_cycle	uint32	
num_std_this_cycle	uint32	
q_rs_unknown_net_check	SIMQ * [MAX_THREADS]	
snoop_cycle	Cycle	

Core related

Name	Size	Description
intransit_count[num_phythread]		表明有哪些RS entry还处在reclaim状态，不能被使用；这个表明某些实现中RS entry的reclaim不能当T完成

Implementation

Flow

Schedule pipeline flow

Dispatch()

主要完成将allocate模块发送过来的uop dispatch到不同的scheduler中，同时处理load/sta(d)间的wakeup关系以及l2的wakeup，回收RS entry资源，决定从哪里进行dispatch(是否使用replay_loop或是从mob进行dispatch)

- 对于enable rob_port_read，表明在allocate阶段是否需要读取ROB/RRF的寄存器，则设置某个ROB entry complete的标志(表明ROB entry的value是最新的)，处理的方式有两种： writeback阶段设置或是dispatch uop的时候设置 （setting_set_arf_at_writeback[1]）
- 对于enable ld_depend_std，表明某个load需要某笔store的数据，并且std wakeup ld需要一定的时间，在这里检查；在mob中local replay load，原因是load需要store的数据
- 对于enable ld_depend_sta，表明某个load被某笔store的地址挡住，并且sta wakeup ld需要一定的时间；在mob中local replay load，原因是load被store的地址挡住
- Unified-L2的处理，对于从L2返回的数据，唤醒mob中的ld uop
- 对于rs_unknown_net_check的情况的处理
- 在当前T起始，清除scheduler的uop和mob_uop的分配计数器为0
- 回收上一个T已经retire的rs entries
- 如果enable setting_replay_loop (P4 feature)，则首先进行replay dispatch
- 如果enable setting_schedule_from_mob，那么执行Mob的schedule
- 执行当T的scheduler scheduler uop流程
- schedule结束后，检查是否对于某些uop需要rebind execport
- schedule结束后，检查损失的带宽？？？

Dispatch_scheduler()

真正的dispatch的逻辑

- 如果setting_allow_sync_mob_rs_dispatch(对于相同的port，mob/rs可以在T都进行dispatch)， 清除dispatch_port_busy
- 当dispatch uop的个数 < scheduler的读口资源，进行uop调度，初始的dispatch uop个数=setting_allow_sync_mob_rs_dispatch ? 0 : mob_ops_scheduled[snum]

allocate阶段分配RS entry(如果不使用UAQ)

首先确定根据uop的execclass确定uop预分配的execport和scheduler

统计信息

mob_uops_scheduled[sched]
uops_scheduled[sched]

scheduler_rate[sched]
scheduler_size[sched]
scheduler_assoc[sched]

scheduler_ptr[sched][phytid][total_uop[sched]] = rid

表示uop进入了sched中，且位于total_uop[sched]的位置，这个uop每次都会尝试去进行schedule，[可能相关的dependency还没有break]

scheduler_tid_ptr[sched][totoal_uop[sched]] = phytid

表示当前处于sched中的uop都属于哪些phytid

scheduler_active_phytid[sched] exclusive with scheduler_tid_ptr

表示当前的sched被哪个phytid占据使用

scheduler_{unfused_}active_entries[sched][phytid]

表示{非fused}的uop由当前phytid在当前的sched分配的个数

scheduler_{unfused_}total_active_entries[sched]

表示{非fused}的uop在当前sched分配的个数

unified_scheduler_{unfused_}active_entries[phytid]

表示{非fused}的uop由当前phytid分配的个数

lost_opportunity_score[port][sched]
dispatch_port_busy[port][sched]

port的端口被哪个sched占用

scheduler_write_port_used[sched]

sched的写口已经被使用了几个

occupied_exec_port[port][sched]

统计信息?

q_delay_sta_wakeup_load_SIMQ
q_delay_std_wakeup_load_SIMQ
q_rs_unknown_net_check_SIMQ[phytid]
q_set_arf_bit_SIMQ[phytid]

问题：

1. Mob_scheduler / scheduler同时调度时对应的HW的结构是什么样子
2. Split exec和sched的HW结构的样子
3. Safe_replay的含义?

<u>与schedule相关的ROB的域</u>

Schedulable	uop可以调度
in_scheduler	uop在scheduler中
extratiny_op/extratiny_op_move	一种特殊的uop，需要额外check
sched_from_mob	当前的uop由mob scheduler进行调度
dependents_not_scheduled	表明当前uop的所有depend后继者不能进行schedule，在rs_check_unknown_sta中进行设置

was_ready_but_blocked_this_cycle	当前uop已经满足了schedule的ready条件，但是因为资源的问题，无法进行schedule
cycle_scheduled	表示uop被scheduler调度执行的时间，在dispatch_uop->update_dispatch_stat中设置
cycle_exec_known_bad	执行阶段exec设置，当uop在执行过程中发生了replay的情况，那么当前uop的执行结果为不可信状态(bad)，对于没有配置replay loop q的情况， 设置此域，后续依赖这条Uop的指令会检查到自己的operand什么时候ready，在dispatch流水线上执行的时候会被kill exec。value = cur_time + uop_latency - sched_clock，反映当前uop什么时候value变为不可信
cycle_result_safe	表示uop的结果是可信的，在exec中设置，表明uop已经执行，cycle_result_safe = cur_time + uop_latency + replay_latency - sched_latency，表示uop产生的结果何时可用，此时还没有写回到regfile，仅仅是执行单元产生了结果
cycle_result_ready	表示uop的结果已经ready，可以唤醒后续依赖的指令 进行schedule，当指令已经确定可以进行dispatch的时候，进行计算和设置，dispatch_uop之前设置(dispatch_update_scheduled_uop_timestamps)，cycle_result_ready = cur_time + uop_latency
? cycle_result_ready_2nd_sched	与recent_load_dep_vector有关系，需要进一步的check， 设置同cycle_result_ready
cycle_exec_completed	表明uop何时执行完有结果，可以进行writeport的回写 (= cur_time + uop_latency + rs_to_stack_latency + {dispatch_latency - sched_latency} [split_sched/exec])
last_dependency	进行uop的src的ready检查的时候，如果发现了 某个src不ready，则将该src的cycle_result_ready标记到这个域
last_dep_adjst	依赖的修正值，目前设置为0

ready的条件

TBD

schedule的条件

TBD

uop latency的计算

这个计算很关键，直接决定了哪些情况需要进行replay，且uop的ready/safe等时间

1. scheduler_adjust_load_latency(rid, phytid); 在uop开始进行schedule的条件检查的时候
2. compute_uop_latency(phytid, s, r); 在uop所有的src已经ready，但因为资源限制无法dispatch

schedule的执行流程

No main flow

1. 更新ROB中的arf bit，通知allocate当前ROB entry中的value已经回写
2. ? Delay std wakeup loads ? 何时发生
3. ? Delay sta wakeup loads ? 何时发生
4. ? Fb_bypass_ul2 wakeup uop ? 何时发生，需要check cache system
5. ? Uop dispatch前需要check unknown sta ? 当load uop已经满足了ready条件，且配置了这个选项，则需要进行unknown sta check，通过SIMQ实现，说明有latency

Main flow

1. 回收sched单元，计算当前T的可用的sched单元个数，释放的单元通过q_intransit_rs SIMQ回收
2. 配置了replay loop q，那么首先尝试从replay q进行schedule
3. 配置了schedule_from_mob（这个不是指ld/st都是从mob进行调度，而是某些特殊情况需要从mob调度)，从mob进行调度
4. 执行scheduler的执行逻辑
   a. 清除当前T的dispatch_busy_port[port][sched]的标志，这个标志说明可能每个port可以有单独的HW逻辑支持每个sched的处理
   b. 遍历每个scheduler，以scheduler的read_port为限制 (每个scheduler最多往exec port发射read_port个uop)，检查每个uop是否可以dispatch到port的逻辑顺序采用从older->yong的uop的allocate顺序，每个sched都有自己的buffer，用来缓存来自于不同phythread的uop

      schedule_uop的流程

      i. 排除一些不能调度的情况
         1) uop不在scheduler中(也不是来自于mob_schedule)，或是处于!schedulable的状态
         2) uop已经执行(split sched/exec)，但是还没有执行完毕， 针对cycle_exec_known_bad > cur_time
         3) ? load，但是dl1当前T做了refill的动作，需要check ld port的个数
         4) ? load，但是dl1当前T执行了snoop的动作，需要check ld port的个数
         5) 对于!safe_schedule和!split_sched/exexc的情况，如果当前已经schedule的各种类型的uop数量已超过了threshold
         6) ? 对于一些特殊情况，uop不用考虑src是否ready （extratiny uop)
         7) uop正在从ROB/RRF中读取一些操作数，这个过程还没有完成
         8) split_sched/exec，且cycle_scheduled != cycle_exec_known_bad，表明当前的uop还没有执行完毕
      ii. 进行ready条件的检查 (操作数是否已经ready)
         1) 所有src都是ready的，如果任意一个不是ready的，那么需要评估是否需要更换执行单元(check_for_rebind)
            a) 对于bogus uop，如果设置了drain_scheds_on_beuflush，那么强制uop已经ready
            b) 对于extratiny uop，视为ready
            c) 对于enable了xarg_mfp的setting，需要检查的src到RET_DEP，否则到STD_DEP，都不包括

               对于每个src
               i) 如果当前src=STD_DEP, 且mfp预测错误，那么修改STD_DEP的依赖为mfp预测的结果
               ii) 如果不依赖或者src=STA_DEP && !sta_specific_prediction，ready
               iii) uop是load，设置了schedule_from_mob，且src=STD_DEP/sched_ld_past_std，或是src=STA_DEP/sched_ld_past_sta，ready
               iv) 计算uop与依赖的uop之间的extra_delay，包括interstack forwarding delay， ld/st cam schedule delay
               ? Cam scheduler/matrix scheduler什么意思，需要找下专利
               v) 判断依赖的uop的结果 (|)

1st. In uaq，不在scheduler中

2nd. !split_sched/exec, 在scheduler中，没有dispatch到execport

3rd. 结果已经ready, cycle_result_ready，但是不满足cycle_result_ready+extra_delay <= cur_time

4th. 结果还没有ready，cycle_result_ready = 0

5th. uop已经设置cycle_exec_known_bad < cur_time，说明uop或其依赖的uop出现了replay情况，且目前不是bogus uop，说明操作数的结果不可信

6th. 对于uop是fp类型的操作

First. Wait condition （|）

1. fp操作需要等待所有src都是ready，设置fp_ops_wait_srcs_safe
2. 依赖的uop是fp_load，uop需要等待fp_load的result是safe的,cycle_result_safe

Second. Timing condition （|）

1. 依赖的uop的结果不是safe的，cycle_result_safe = -1
2. result是safe，但是cycle_result_safe + extra_delay > cur_time + replay_latency

当出现上面的情况时

1st. uop标记依赖的uop的cycle_result_ready

2nd. 对于src=STD_DEP的情况，如果mfp预测ld/std没有dependent，那么标记当前uop.std_not_ready=true，并且设置ld.waiting_for_int_std(allocate阶段总是标记fwd_from_int，这个也是有点疑问)

3rd. 更新当前uop不能schedule的原因为dep_uop的原因(no_sched_reason)，或者如果dep_uop是cache miss的程序(ul2_miss)，那么设置当前uop为WAITMISS

4th. 对于WAITMISS的uop，如果设置了remove_from_rs的setting，那么将当前uop从sched中移除

对于没有出现上述情况的dep_uop(src=STD_DEP)，则当前dep_uop是ready。特别地，如果mfp没有预测std/ld间的依赖关系，那么设置std_was_ready

2) 或者，之前已经check过了，且已经找到了一个不满足条件的dep_rob，如果当前dep_rob的dst操作数还不满足。这里的不满足表示cycle_result_ready=0或是cycle_result_ready+dep_adjst(0) > cur_time　　问题：在HW上是否必须？？

iii. 进行schedule条件的检查 (资源和语义的问题)

1) scheduler_find_free_execport——allocate阶段已经进行了port binding，或者schedule前进行port binding，如果当前的port已经被占用了，那么当前uop没有port可以执行

2) scheduler_wb_port_stall——调度时writeback port存在冲突，考虑是否进行port rebind

3) 设置了safe_schedule的setting，那么uop必须等到所有依赖的src的值是safe的（这里仅仅考虑到STD_DEP之前），才能进行调度；当本满足safe的调度条件，那么检查当前的uop是否等待miss，如果是，如果设置了remove_uop_waitmiss_rs，那么将当前uop从rs中移除

a) Dep_uop不是bogus uop

b) Dep_uop不是extratiny uop

c) 对于uop的各个src

i) 不check load/std之间的dependency

ii) load/sta dependency, 但是配置为sched_ld_past_sta，那么不check

iii) 配置ignore_i2a(integer2address)_dep_sched，那么如果dep_uop属于general_sched，uop属于memory_sched，不check

iv) Dep_uop的cycle_result_safe为-1(表明当前的dep_uop还没有执行)，或者uop_timestamp(sched: cur_time+replay_latency; cur_time) < dep_uop.cycle_result_safe + fp_inter_latency(fp clusters delay, no fp cluster) + extra_latency(0)

4) 对于load来说，需要check与之依赖的sta是否符合schedule条件

a) 当前uop是load，且没有设置sched_ld_past_sta

b) 当前uop不是bogus uop

c) 当前uop设置了dependent_on_older_stas的标志，这个标志表示 (|)：

i) mfp不做sta/ld间的预测(!mfp_sta)，则ld依赖于前面的所有的load——这里仅仅是标注，需要在schedule的时候进行检查

ii) mfp做了std/ld间的forwarding的预测，且ld的STA_DEP有值，且!sta_specific_prediction

iii) mfp没有找到std/ld间的依赖，但是发现了一个独立的sta与ld有dependent，且!sta_specific_prediction

d) 如果load访问的是stack，那么!sta_no_sched_guarding_ss

如果当前ld/std之间有forwarding关系，那么增加st_ld_forwarding_delay(sta_ld_fwd_penalty) [问题：这个penalty对应的HW的设计是什么样子？] check的方法：

逐一扫描每个ld之前的sta{扫描的方式有些奇怪——沿着speculative_context的链进行扫描，是否对于一个phythread而言，可能为不同的thread分配不同的speculative context路径？？}

i) sta已经获得结果，但是cur_time < cycle_result_ready + st_ld_forwarding_penalty

ii) sta还没有执行获得结果cycle_result_ready = 0

iii) sta还不是bogus uop，但是cycle_exec_known_bad < cur_time——表明sta的exec被cancel，等待再次被执行

标记当前schedule的load再次schedule的时候需要先检查依赖的sta的完成性 (转到 上面2)

5) slow_lock的条件不满足，lock_load必须是完全串行执行(backend empty, 且所有的store已经global observed)；当上面条件满足后，则lock_load需要满足调度的schedule_latency = cur_time + 4*replay_latency

6) slow div unit的条件不满足，后续具体看(dispatch_divide_stall)

7) 如果所有的load需要in-order dispatch——本条件没有支持

对于ready不能schedule的uop，会进行标记was_ready_but_blocked_this_cycle，在后续的步骤6中会进行counter的update

iv. 对于在allocate阶段进行port binding，且不是从mob schedule，且(!pb_dec_at_reclaim || ! split_sched/exec)，进行port上counter的更新，并设置uop.dec_inhibit

v. 如果uop是个load，且设置了mdisamb_enable，且不是从mob进行调度，检查mdisamb_table，如果hit table且饱和计数器>预设值，那么设置

uop.mdisamb_allowed = false，用于后续的store/load之间的check

vi. 如果设置了replay loopq，或是!split_sched/exec，那么把当前可以schedule的uop从scheduler中移除

vii. 记录uop之前的schedule情况，包括port/scheduler/is_mob_schedule

viii. 更新schedule_uop的timestamp，这里主要是指cycle_result_ready / set !schedulable / set !in_scheduler

1) 对于load且不是从mob_schedule的情况，如果setting rs_dispatch_check_sta_unknown，那么加入q_rs_unknown_net_check的SIMQ中，进行check
2) 对于split的请求，标记cycle_result_ready=0, TBD
3) 设置uop的cycle_result_ready标记，表示后续的依赖指令可以继续调度
4) 设置uop为不可调度状态(schedulable = false)，如果!split_sched/exec，那么标志当前uop已经移除RS

ix. 进行uop的dispatch

5. 检查因为已知的writeback port冲突，需要进行port重新分配的情况
6. 检查已经满足ready条件，但是无法schedule的uop，更新一些管理信息 [bias_matrix / lost_opportunity_score{感觉没有使用，或者代码逻辑问题}]，用于后续uop的port bind的优化

scheduler不能schedule的若干条件

1. SCHED_OK
2. NODISPPORT
3. NOWBPORT
4. NOFU
5. NOLDPORT
6. NOSTAPORT
7. NOSTDPORT
8. WAITFORMISS
9. SRCF_ROB_READ
10. ROB_READ

dispatch的执行流程——将uop从scheduler送到port的过程

1. Enable_rob_read_port且!set_arf_at_writeback，那么将对应的rob entry加入到q_set_arf_bit SIMQ中
2. 设置dispatch_port_busy[port][sched]的状态位，如果no_sched_port_limitation setting，那么不设置
3. 如果full_portbinding setting，那么所有的scheduler共享port的dispatch_port_busy标志
4. 更新scheduler dispatch的统计信息，这个信息会在scheduler的第i.5中使用，且设置cycle_schedule，表明uop在cur_time进行dispatch
5. 对于首次执行的uncachable的uop，需要invalidate dl1/ul2 ??
6. **!split_sched/exec，处理exec的replay条件**
   a. **进行replay的check**

   b. **对于设置了schedule_from_mob，且uop是ld/st，且存在replay的情况**
      i. **对于bogus uop的情况 ---> 会有这种情况吗？前面应该已经进行了bogus uop的处理**
         1) **如果是个split访问，那么释放split register**
         2) **如果当前uop占用了pmh，那么释放pmh**
         3) **对于uop的miss情况，如果预留了fill buffer，那么释放掉reserved_fb(fp_reserved_release)**
         4) **如果当前uop没有设置cycle_result_safe，设置cycle_result_safe = cur_time + uop_latency + replay_latency**
         5) **对于！Instant_reclaim_bogus_buffers的情况，则将uop push进入q_retire的SIMQ，等待进行release**
      ii. **不是bogus uop的情况，直接放入到mob scheduler中**
7. 处理dispatch的replay条件

   实际上并不是处理dispatch的replay条件，而是进行writeport reserved / idiv reservation的一些处理，如果不能满足，需要设置一定的replay条件，但是对于split_sched/exec的结构，dispatch不会出现replay

   a. 设置cycle_exec_completed，这个用来计算writeport的占用情况
   b. 如果设置了writeback_ports_enable，那么dispatch的时候reserved writeback port
   c. 如果schedule的时候已经设置了idiv unit的预留，或者当前进行idiv unit调度没有stall(dispatch_divide_stall)，那么更新idiv的资源表示(scheduler_update_divide_resources)
      i. 否则，必须设置了setting_replay_loop，则设置replay条件为REPLAY_CAUSE_DIV_BUSY

   对于模拟器来说，对于replay的HW支持结构如下

   a. !split_sched/exec
   b. Split_sched/exec，且enable replay loopq
   c. Split_sched/exec，且没有replay loopq，那么不能出现replay的条件
8. 检查当前dispatch的uop的src是否已经cycle_exec_known_bad
9. 处理recent_load_dep_vector，？？
10. 如果setting ld_past_std，且当前dispatch uop为std，且setting schedule_from_mob，且!wakeup_ld_on_non_spec_store || !cycle_exec_known_bad，std需要wakeup和它有依赖关系的load，如果wakeup_ld_on_non_spec_store为1，为什么uop的cycle_exec_known_bad为0就代表了non spec store的处理?
    a. !delay_std_wakeup_of_loads || mob_wakeup_ld_on_std_dispatch，在std dispatch的时候直接唤醒mob中等待的load
    b. 否则，!split_sched/exec，将当前std加入q_delay_std_wakeup_load的SIMQ中，用于唤醒mob中等待的load
11. 对于sta和load之间的依赖关系，如果split_sched/exec，且!delay_std_wakeup_of_load，且ld_past_sta且当前是sta，且schedule_from_mob，且!wakeup_ld_on_non_spec_store || !cycle_exec_known_bad，在sta dispatch的时候直接唤醒mob中等待的load，如果wakeup_ld_on_non_spec_store为1，为什么uop的cycle_exec_known_bad为0就代表了non spec store的处理?
12. !split_sched/exec，且不存在replay，那么execute_uop
    a. !br && opcode!=invalid_page，reset btflush，什么目的？？
    b. 处理branch mis-predict (core_mispredicted)

i. 设置cycle_beuflush为当前cur_time
ii. uop不是bogus uop，且instant_reclaim_bogus_buffers，回收上次使用的branch_color，用于branch checkpoint的处理
iii. 设置pending_beuflush[tid]标志
iv. 如果是stall_until_mispred_retire的方式，则设置stall_on_branch_rid[tid]
v. 如果是bpu_mispredict
1) 进行phytid的thread switch，如果setting phythread_sched_policy == E_T_MISPRED_POLICY，尝试切换phythread
2) Clean_window，针对当前pipeline上和buffer中的uop进行清除
3) 进行periodic_checkpoint或是br_checkpoint两种branch flush方式的资源回收，具体可以看allocate模块
4) 进行frontend reset & refetch，使用q_beuflush SIMQ
vi. 如果!update_bp_at_retire，则准备进行bp table的更新，通过q_bp_update SIMQ

c. 设置cycle_result_safe = cur_time + uop_latency + replay_latency

13. !split_sched/exec，且uop之前调度过(cycle_scheduled!=0)，且没有replay，清除idiv执行部件的占用状态
14. !schedule_from_mob || (!is_load && !is_sta) || no_replay
进行lock_load的处理，且uop不是bogus uop
a. (!split_sched/exec || "oldest" uop) && do_seriesold_base_locks
设置mob_load_lock_sched[tid]/mob_load_lock_paddr[tid]，设置lock_asserted
b. do_CPU1_fast_locks && uop.speculative_lock(在allocate阶段进行设置)
标记当前phythread speculative_lock_in_progress
15. 如果cur_time == cycle_exec_known_bad，表明当前uop已经知道src是不可信的，那么直接cancel本次的执行
a. TBD
16. 否则，将uop加入到对应的执行队列中 q_dispatch_rid SIMQ，这个可能仅仅是个模拟用的queue
17. 对于!split_sched/exec且replay_loop且has_replay，统计replay的情况

execute的执行流程

TBD

schedule需要考虑的情况

1. replay情况 (dispatch_handle_replay)

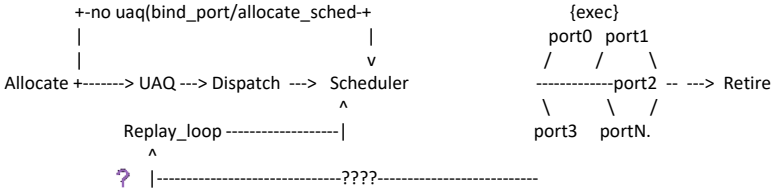
execute需要考虑的情况

1. replay情况 (exec_handle_replay)

OoO的流水线结构

Cancel_exec的流水线情况

ReplayQ的流水线情况



mob_schedule???