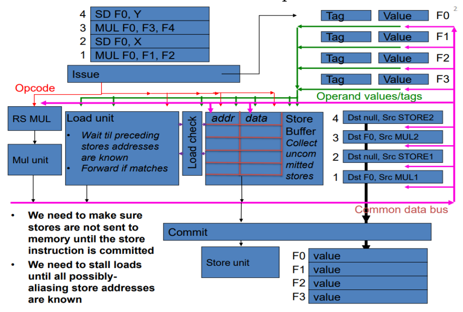
AT23 Advanced Computer Architecture (COMP60001 @ Imperial College London) by ck21

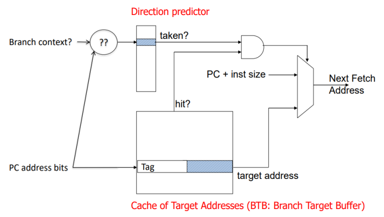
Dynamic Scheduling & Speculation

Pipeline Hazard. Allow instr behind stall to proceed. Data Dependence between Instr - **RAW hzd**. Name Dependence (two instr use same reg/mem loc but no flow of data between instr) - **WAR hzd**. $Instr_1$ writes operand before $Instr_2$ writes it - **WAW hzd**.
Tomasulo Algo. Issue - get instr from FP Op Queue. If RS free (no struct hzd), ctrl issues instr and sends operands (renaming regs). Execute - operate on operands (EX). When both commands ready then ex.; if not ready, watch CDB for result. Write - finish execution (WB). Write on CDB to all awaiting units. Mark RS available. Drawbacks - Complexity, Performance limited by CDB, Non-precise interrupts. T.A. can overlap iterations of loops because of register renaming (multiple iterations use diff. physical destinations for registers - dynamic loop unrolling) and RS (permit instr issue to advance past int ctrl flow ops; buffer old values of reg avoiding WAR stall).
Spec Exec. Need to fix OOO completion to find precise breakpoint in instr stream. Add stage that *commits* state in issue order. 3rd Stage - write to Reorder Buffer (ROB). 4th Stage - update reg with reorder result. When instr at head of ROB, and result present, update the commit side reg (CSR) with the result or store to mem, and remove instr from ROB. E.g. - Br predicted not taken. When BEQ reaches head of commit q, all instr issued not yet committed are erroneous. All ROB trashed, issue side registers reset from CSR. Correct BT instr fetched and queued.
Store Buffer. Make sure STR not sent to memory until STR instr committed. Stall LD until all preceding STR committed. If addr of LD and preceding uncommitted STR are known - none of STR address matches LD, then LD can proceed. If LD Addr matches Addr of uncommitted STR - fwd STR data to LD.
Store-to-Load Forwarding. LD and STR uses computed addresses - may not be known as issue time. E.g. i_1: SD F0 0(R3), i_4: LD F2 0(R3). Speculate forward and check misprediction.
RUU vs ROB. In Tomasulo/ROB registers and ROB entries have a tag. Every register, ROB Entry and RS needs comparator to monitor ROB. RUU → Tags are ROB entry numbers. ROB entry serves as renamed register for instr result. When instr completes, need to check whether ROB operands match.



Branch Prediction (Direction, Target)

Why BP? Amadahl's Law → Impact of ctrl stalls larger with lower CPI in n-issue processor. Spec Dynamic Instr Scheduling with RR enables many instr to be speculated. BP alternatives - multiple threads per core (BEQ → thread 0), predicated execution (turn branches into conditionally executed instr: p1, $< p1 >$ LDR r1 - do i_2 if i_1), delayed branch - br takes place after follow instr (i.e. X instr is executed regardless). Need to predict conditional branches (direction prediction) and indirect branches (target prediction).
BP Schemes. Branch History Table (BHT). 1-bit values at lower bits of PC addr index table → br taken or not last time. Possible aliasing - 2 diff br instr map to same BHT entry. 1-bit BHT cause 2 mispred - End of Loop, First Time through Loop. Solution - Dynamic Br Prediction (adds *Hysteresis*). Generalises to n-bit BHT (Bimodal Predictor) - works well because most br highly biased. Either almost always taken or almost always not. Can use Global History (m most recently-executed br). Keep m-bit Br Hist Reg (BHR) - Shift Reg recording taken not-taken direction of last m branches. Popular Gselect → 4 tables each of $2 * 2^k$ bits.
Tournament Predictor. 2 predictors; one local one global and combine with selector. Selector driven by predictor.
Br Target Buffer (BTB). Need address same time as prediction. Cache of BTA accessed in parallel with I-cache in Fetch stage. Updated only by taken br (direction-predictor determines if BTB used). If BTB Hit and Instr is predicted-taken: tgt from BTB used as fetch address in next cycle. If BTB miss or Instr is predicted-not-taken: $PC + N$ used as next fetch address in next cycle. Note: Disable WB and Mem on misprediction.

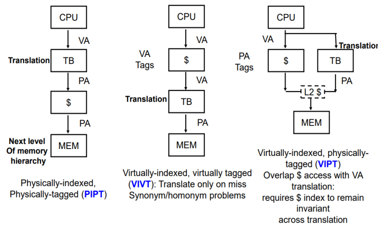


Combine Fast-Simple with Slow-Bigger Predictor. If slow pred differs, re-steer: squash first pred and fetch from improved pred.
Return Address Predictor. Function may be called from different places. It must return to the right place. Address of next instr must be saved and restored. JSR must save RA somewhere. x86 → pushes RA onto stack. RET jumps to address on top of the stack. MIPS → JAL F jumps to F but stashes current PC in special register $\$ra$. Function returns with indirect jump $jr\ \$ra$. If func body has other calls, compiler pushes $\$ra$ to stack. So, we keep small HW stack in BP that mirrors program call-return stack. Value at top of stack used as pred next PC when BTB predicts current instr is RET. If call stack deeper than RAP stack → RAP stack empty. Prediction from RAP may be wrong → RA overwritten, SP changed, thread switch.

Caches & Improving AMAT

Misses. 4Cs - **Compulsory** (First access), **Capacity** (If cache cannot contain all blocks needed), **Conflict** (Collision miss - Block discarded and later retrieved if too many map to its set), **Coherence** (Data invaidated by another processor or IO device).
Cache Anatomy. Block/Line - unit of allocation. Tag - mem addr of cached block. Set - set of cache blocks indexed by given cache index. Way - set of alt locations for stored block in given set, Comparator - check tag matches addres, Selector - picks data from the way with matching tag.
Reduce Miss Rate: Associativity Conflict Misses. ↓ Miss Rate ↔ ↑ Associativity ↔ ↑ Cache Hit Time. To combine fast hit time of DM yet avoid conflict misses: Victim Cache → Add buffer to place data discarded from cache. On miss, allocate into DM cache. On replacement, allocate into victim cache. On access, check both. On victim cache, re-allocate into DM cache. Commonly used for last-level caches. So, reduce impact of misses by providing a small, fast cache specifically designed to hold evicted lines.
Skewed-Associative Caches. In conventional w-way set-associative cache, get conflicts when $n + 1$ blocks have same address index bits. Reduce conflict misses by using different indices in each cache way. Introduce simple hash func → XOR index bits with tag bits and reorder index bits. Costs: One address decoder per way, latency of hash, difficulty of LRU.
HW Prefetching. Extra block placed in stream buffer. After cache miss, stream buffer initiates fetch for next block. But not allocated into cache to avoid pollution. On access, check stream buffer in parallel with cache. Relies on having extra mem bandwidth. Extend to Multi-Way → Track multiple access streams simultaneously; one stream good for instr-cache misses. Important for data (traverse multiple array).
Decoupled Access-Execute. Separate instructions that generate addresses from instructions that use memory results; address-generation runs ahead.
Software Prefetching. Trigger prefetching explicitly → addresses that would cause page fault or protection violation. Silently squash. We can also reduce Instr-cache misses. Storage Layout - Merge Arrays (Improve SL by single array of compound elements vs 2 arrays. 2 seq array → 1 array of structs), Permuting Multidimensional Array (Improve SL - match array layout to traversal order); Iteration Space - Loop Interchange (Change nesting of loops to access data in order stored in mem), Loop Fusion (Combine 2 indep. loops that have same looping and overlapping vars), Blocking (Improve TL by accessing blocks of data repeatedly vs going down whole cols or rows).
Reduce Miss Rate Penalty: WT vs WB. WT: All writes update cache and underlying mem/cache. Can always discard cached data. Most up-to-date data is in memory. Cache control bit: Valid bit. Mem always has updated data; simpler mgmt. WB: All writes simply update cache. Can't just discard cached data. May have to write it back to memory. Cache control bit: Valid and Dirty Bit. Lower B/W, Better Tolerance to long-latency memory.
Write Allocate and Non-Allocate. Simply send write data through to underlying memory/cache. Don't allocate new cache line.
Read Priority over Write on Miss. WT with Write Buffers. RAW conflicts with main mem reads on cache misses. Sol: Check write buffer contents before read; if no conflicts, let mem access continue. If we use write-back, also need write buffer to hold displaced blocks. Read miss replacing dirty block. Normal - write dirty block to mem and then do read. Instead, copy dirty block to write buffer then do read and then write.
Early Restart and Critical Word First. Processor can continue as soon as the requested word arrives. Don't wait for a full block. ER - as soon as the requested word of the block arrives, send to CPU and let CPU continue. CWF - Request the missed word first from memory and send it to the CPU as soon as it arrives. Generally useful only in large blocks. We can also sector the cache lines; each with validity bits. Allocate in units of cache lines but deliver data in units of sectors. Fetch sectors in any order, even leaving unvalid until requested.
Non-Blocking Caches to Reduce Stalls on Misses. Allows data cache to continue to supply cache hits during miss. Hit under Miss - reduces effective miss penalty by working during miss instead of ignoring CPU requests. Hit under Multiple Miss / Miss under Miss - overlap multiple miss to reduce miss penalty. Increase complexity of cache controller (outstanding memory accesses).
What happens on miss? Freeze Pipeline in Mem Stage; Use Full/Empty Bits in Registers and MSHR (Miss Status / Handler Registers) Queue (Each Entry keeps track of status of o/s memory

req in one complete mem line).
Multilevel Inclusion. L_{n+1} contains everything in L_n. Issues - Replace dirty lines and cache coherency (invalidation). MLI - If line not in L2 don't need to invalidate in L1.
Fast Cache Hits by Avoiding Translation. If the cache index consists only of phy bits of the address, we can start tag access in parallel with translation so we can compare to phy tag. But this limits cache to page size → what if I want bigger caches and still use the same trick? (1) Higher Associativity, (2) Page Colouring (Cache Conflict when 2 cache block with same PA (tag) mapped to two different VA. Make sure OS never creates page table mapping with this property → Synonym Consistency) So - take Addr Trans off critical path; access TLB in parallel with L1 cache (do not use translated bits as index bits); TLB is cache of Addr Trans.


CPU
VA
Translation
TB
PA
$
PA
Next level Of memory hierarchy
MEM
Physically-indexed, Physically-tagged (PIPT)
CPU
VA
VA Tags
$
VA
Translation
TB
PA
MEM
Virtually-indexed, virtually tagged (VIVT): Translate only on miss Synonym/homonym problems
CPU
VA
Translation
PA Tags
$
TB
PA
L2 $
MEM
Virtually-indexed, physically-tagged (VIPT)
Overlap $ access with VA translation:
requires $ index to remain invariant
across translation

Sidechannel Vulnerabilities

Exfiltration via Shared States. Prime and Probe: Reveal shared cache states via probing and checking cache access times. Evict and Time: Measure variations in execution times after evicting specific line. Flush and Reload: Flush shared VirtMem, measure time to reload to detect victim access.
Spectre Variant 1. Bounds check predicted satisfied but i out of bounds. Speculatively use s as an index into an array that we do have access to. Use timing to determine whether cache line on which B[s] falls has been allocated as side-effect of speculative execution. Extension: Speculative accesses can also be made to addresses in kernel memory (Spectre allows access to OS secrets / other user processes → Meltdown → KASLR and KPTI (Kernel Address Space Isolation) as Mitigation).
Spectre Variant 2. Accessing data in different address space → trick victim into accessing data we want. Gadget → persuade kernel to jump to label (has secret). We can prime BTB to jump to gadget. Find gadget, train BP to cause spec branch to gadget when syscall executed. Observe microarchitectural or cache side channel from spec-exec gadget. Steal secret. Mitigations - block side channels, mess with cache probing (noise), prevent attacker from poisoning BP, block BP contention.
Retpoline. Code sequence that implements indirect branch using return instruction. Fixes return address stack to ensure benign prediction target. Prevents processor from unsafe speculative execution. RP0: call RP2; RP1: int 3 (breakpoint interrupt → safe loc); RP2: mov [rsp], $< JumpTgt >$; RP3: ret

Software Pipelining & VLIW

Loop Unrolling. Replicate loop body multiple times, reducing overhead of loop control instructions. Cf. Software Pipelining, where the execution of multiple iterations of a loop are overlapped. S.D. 0(R1) F4; ADD.D F4,F0,F2; LD F0,-16(R1). So, fill and drain once per loop vs once per unrolled iteration in loop unrolling. ↑ ILP, Tomasulo in SW.
VLIW. Each instr has explicit coding for multiple ops. IA64 - grouping called packet. All ops compiler puts in the long instr word are independent (issued and executed in parallel). Instruction group: seq of consecutive instr with no reg data dependencies. Arbitrarily long but compiler but explicitly indicate boundary → stop.

Simultaneous Multithreading & SIMD

Intro and Issues. SMT: Max Util of Function units by independent ops. Dynamic Scheduling of ops from pool of threads. Each thread may run slow - but point of SMT is that resources are dynamically assigned. If only one thread it can run faster. But SMT threads contend for resources. They need to be scheduled fairly (one thread cannot monopolise CPU). Side Channels. SMT threads need to exploit mem-system parallelism (get mem access in flight).
AVX512 Vector Addition. instr executed in parallel across lanes with rich set of instructions that operate on 512b operands. In C, Compiler provides vendor intrinsics that can emit specific vendor instructions. Only lanes with corresponding bit set in predicate register $k1$ are activated. Two predication modes: masking and zero-masking. Former - inactive lanes do not overwrite prior contents. Latter - inactive lanes produce zero.
Explicit Vectorisation: Ignore Vector Dependencies. `#pragma ivdep` Compiler hint - assume no loop-carried dependencies (vectorisation is safe; but it still might not vectorise). Also, OpenMP (`#pragma omp simd`) - loop can be transformed into SIMD loop or declare simd → enable SIMD instr at function level from SIMD loop, SIMD intrinsics (specific vector instructions), SIMT (vectorise outer loop).
Alternatives. Vector Pipelining: Simple static pipeline. Vector instr executed serially, element-by-element, using pipelined FU. Several pipelined FUs. Vector Chaining - each word forwarded to next instr. Long pipelined chain.
UOP Decomposition. DySch OOO Machine. N-wide vector instr split into m-wide uops at decode time. DySch execution engine schedules their execution posibly across mul FUs. Committed Together.
SIMD Architectures. Lane-by-lane predictaion allows conditionals to be vectorised → branch divergence may lead to poor util. Indirections an be vectorised but hard to implement efficiently unless accesses happen to fall on small number of distinct cache lines.
SIMD → SIMT Translation: Expand Scalar Lanewise Instr in SIMT into Vectors (16 Lanes of Data) - SIMD. Tricky if there's if - use predication.

GPUs

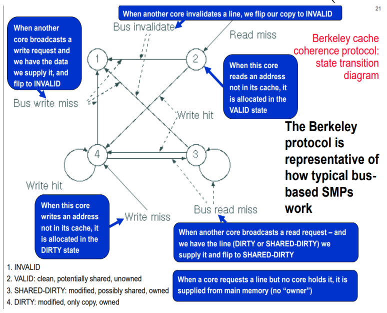
Difference. Never speculate (always another thread waiting with work), no spec br exec (or even BP), FGMT or SMT to hide cache access latency.
CUDA. Parallel GPU Code (C Ext). GPU Kernel is C function. Each thread executes kernel code; group of threads is Thread Block and TB organised into grid. Threads within same TB can sync exec and share access to local scratchpad memory (hierarchy of parallelism). TB allocated dynamically to SMP (≈ multicore).
Warps. New Model: Single Intr Multi Thread (SIMT). SM SIMT unit manages threads in groups of warps. SIMT similar to SIMD, one instr multiple data lanes. Difference: SIMT applies one instr to multiple indep threads in parallel, not just multiple data lanes. SIMT instr controls exec and br behaviour of one thread.
Br Divergence. In warp, threads all take the same path or diverge. A warp serially executes each path, disabling some threads. When all paths complete, the threads reconverge. Divergence only occurs within a warp - different warps execute independently. Ctrl Flow Coherence: When all threads in a warp goes the same way - good util (SL).
SIMD vs SIMT. SIMT - One thread per lane; adj threads (warp) execute in lockstep; SMT so multiple warps run on same core to hide mem latency. SIMD - each thread may include SIMD vector instr; SMT so small number of threads run on the same core to hide mem latency. SIMT - SL adj threads access adj data; LD instr can result in completely different addr accessed by each lane; Coalesced loads (accesses almost adj, run faster). SIMD - SL adj loop iterations access adj data. SIMD vector LD has access to adj locs.

Multicore

MPI. Need to make comms explicit. Single Program Multiple Data (SPMD) - every proc has share of data and same control flow. MPI_Init (Initialise MPI), MPI_Comm_Size (find out how many prcoesses), MPI_Comm_Rank, MPI_Finalize (terminate MPI). Collective Ops: MPI_Bcast (Broadcast data from process with rank root to all other procs in grp), MPI_Reduce (Combine value on all processes into single value using op defined by parameter op (e.g. sum)), MPI_AllReduce (MPI_Reduce then broadcast to every process has sum).

Cache Coherency

Goals. Processors should not continue to use out-of-date data indefinitely. Every LD instr should yield result of most recent STR to address. Sequential Consistency: result of execution is same as if ops of all processors were executed in some seq order, and ops of each indiv proc appear in seq of order specified by its program.
Invalidation and Snooping. Instead of updating remote cache lines, we invalidate them all when STR occurs (see: Berkeley


When another core invalidates a line, we flip our copy to INVALID
21
When another core broadcasts a write request and we have the data we supply it, and flip to INVALID
Bus invalidate
Read miss
Berkeley cache coherence protocol: state transition diagram
1
2
When this core reads an address not in its cache, it is allocated in the VALID state
Bus write miss
Write hit
The Berkeley protocol is representative of how typical bus-based SMPs work
4
3
Write hit
When this core writes an address not in its cache, it is allocated in the DIRTY state
Write miss
Bus read miss
When another core broadcasts a read request – and we have the line (DIRTY or SHARED-DIRTY) we supply it and flip to SHARED-DIRTY
1. INVALID
2. VALID: clean, potentially shared, unowned
3. SHARED-DIRTY: modified, possibly shared, owned
4. DIRTY: modified, only copy, owned
When a core requests a line but no core holds it, it is supplied from main memory (no "owner")

Protocol).
Snooping. Cache Controller implements protocol state transitions, which snoops all bus traffic. Transitions → bus or CPU. For every bus transaction, it looks up the directory (cache line state) for specified addresses. If processor holds the only valid data (DIRTY) response is Bus Read Miss, providing data to requesting CPU. If memory out of date, one of the CPUs will have cache line in S-D state so must provide data to req. CPU.
Synchronization. Need uninterruptible primitive to fetch and update memory (atomic operation). Can build user level synchronisation ops using this primitive (lock unlock, barrier, fetch and add). Direct HW Implementation: Test-and-set (test value and sets it if value passes test), Fetch-and-increment (returns val of mem loc and atomically increments it), Atomic Exchange (interchange a value in reg for value in mem).
Load Linked Store Conditional. LL returns initial value. SC returns 1 if it succeeds (no other STR to same mem loc since LD) and 0 otherwise (no invalidation).
Spin Locks in Multicore. Processor continuously tries to acquire, spinning around a loop trying to get the lock. But EXCH includes write (invalidation) and this generates bus traffic. Sol: Repeatedly read the variable, when it changes then try exchange.
Ticket Locks. Explicitly hand off access to next in line.
Memory Consistency. SeqConsistency: Res of execution same as if accesses of each proc were kept in order and accesses among diff proc interleaved. (assignments before ifs).
Directory, ccNUMA, S3MP. Bus becomes bottleneck when there's many procs. DRAM also distributed (Non Uniform Memory Arch) - each node allocates space from local DRAM with copies of remote data in cache. ccNUMA - each node has fragment of system DRAM, every physical address has unique home node. COMA - each node (NUMA domain) has fragment of system DRAM but data migrated between NUMA domains adapatively. NUCA - distributed cache so access latency nonuniform.