Memory

Latency: (seconds/cycle) time taken for a single operation from start to finish
Bandwidth: (operations/second or operations/cycle) maximum rate at which operations can be performed (maximum throughput)
Occupancy: time during which the unit is blocked on an operation
CPU-Memory Bottleneck: performance typically limited by memory bandwidth & latency (time taken to perform memory access » processor cycle time)
Processor-DRAM Gap: Increasing 50% per year
Physical Size affects Latency: memory access latency increases with bigger chips as signals must travel greater distances
Memory Hierarchy: Registers (Instructions & Data) ↔ Cache/SRAM (Blocks) ↔ Main Memory/DRAM (Pages) ↔ HDD
Capacity: Registers « SRAM « DRAM
Latency: Registers « SRAM « DRAM
Bandwidth: CPU ↔ Registers/Cache » Cache ↔ DRAM
Locality: programs use a relatively small portion of the address space at any instant of time
Temporal Locality: if a location is referenced by a program it is likely to be referenced again in the near future (think of loops)
Spatial Locality: if a location is referenced it is likely that locations near it will be referenced again in the near future (think of array accesses)
Locality of Cache: cache exploits temporal locality by remembering the contents of recently accessed locations (storing them in cache) & spatial locality by fetching blocks of data around recently accessed locations (fetching multiple blocks at a time into cache)

Cache

Caches needed as processor-DRAM gap increasing & DRAM not providing fast access times for modern CPUs
Block/Line: smallest portion of data that can be stored in the cache - determines the granularity of caching (larger blocks reduce overhead of fetching data from main memory due to spatial locality but increase likelihood of replacing useful data & increasing miss rates)
Sets: a group of cache blocks within a cache (in set-associative each set contains a few blocks which are indexed by the same tag)
Ways: represent the number of cache blocks within a single cache set (2-way set-associative: 2 blocks per set as cache split into 2 ways)
Direct-Mapped Cache: one-to-one mapping between data in memory & data in cache - unique location in cache for each memory address
Contains Valid Bit - Cache Tag - Cache Data
Tag used to determine if the data in that block is the requested data
Associativity Conflicts occur as multiple memory locations map to the same single cache block (accesses to memory mapped to the same block can overwrite each other)
Set-Associative Cache: for a particular memory location - multiple places in cache can store the data (one location in each way)
Fully-Associative Cache: one memory block can be mapped to any cache block (access all cache blocks at once & do a tag comparison for every block in cache)
Increasing Associativity: more comparators & higher energy cost - better hit rate (diminishing returns) - reduced storage layout sensitivity (more predictable as less likely two addresses overwrite each other) - smaller tag
Block Replacement:
on a miss in set/fully associative cache - choice of which block to replace
LRU: hard to implement & must track accessing of ways (commonly used)
Random: easy to implement - loses some spatial locality - only beaten by LRU for small caches
Write Strategies:
Write-Through: information written to both the block in the cache & the block in lower-level memory (always combined with write buffers so CPU does not have to wait for lower-level memory)
Write-Back: information written only to block in cache (modified cache block is only written to main memory at some time later, when the cache needs to evict a block of data from the cache) (as absorbs repeated writes - big energy advantage, less often needs to access lower levels of memory hierarchy) (may block writing to higher levels of memory hierarchy - unless a WB buffer used)
WT: reduced complexity - higher memory bus traffic - higher write latency - inefficiency
WB: reduced memory bus traffic - lower write latency - aggregates writes - efficiency - increased complexity - cache coherency issues
On Cache Miss: No Write Allocate (write to main memory) + WT / Write Allocate (fetch into cache) +WB
Cache Performance: AMAT = Hit Time + Miss Rate x Miss Penalty (improve by reducing hit time - reducing miss rate - reducing miss penalty)
Cache Misses: Compulsory (first reference to a line) - Capacity (cache too small to hold working set of program) - Conflict (misses that occur due to collisions)
Larger Cache Size: reduces capacity & conflict misses (conflict misses reduce as you have more blocks - so more unique tags - and less memory locations map to the same tag) - hit time increases (physical distance to data in cache increases)
Higher Associativity: reduces conflict misses (more places to store data with the same tag → more blocks in a set) - may increase hit time (more comparators to check for hits)
Larger Block Size: reduces compulsory misses (more data loaded into cache at once → filling up the cache faster) - increases conflict misses & miss penalty (increases conflict misses due to fewer sets in cache & fewer tags - more opportunities for conflicts) (higher miss penalty as more data to be reloaded on miss)

Shared vs Distributed Memory(1)

Shared Memory: → allows multiple processors to access the same memory space → makes parallel programming easier as all processors can directly read & write to the same memory locations without the need for complex message-passing protocols (needed in distributed memory systems) → need to ensure cache coherence → locks, semaphores, & barriers used to avoid race conditions when processors accessing the same shared memory
Distributed Memory: each processor has its own private memory → processors communicate through message passing → introduces communication overhead → more complex programming needed - as requires explicit management of synchronisation & communication
Shared vs Distributed: ease-of-use → shared memory is typically easier to program due to its straightforward memory model (but can become complex for a large number of processors due to cache coherence issues) - scalability → distributed memory scales better but requires careful management of communication & synchronisation - suitability → shared memory is suitable for small to medium-scale parallelism (i.e. multicore processors) while distributed memory is preferred for large-scale parallelism (i.e. computer clusters) - performance → shared memory is faster as can communicate with just a load store (no message passing overhead - as in distributed)

Shared Memory Parallel (OpenMP): application programming interface (API) for shared-memory parallel programming - typically found in multicore processors
Features:
Directive-Based Model → OpenMP used compiler directives (pragmas) that tell the compiler how to parallelise the code → allows for incremental parallelisation (can begin with a serial program parallelise step by step)
Synchronisation Mechanisms → contains mechanisms like barriers, critical sections, locks, & atomic operations to synchronise execution of threads (avoiding race conditions)
Data Environment → provides mechanisms for specifying how variables should be treated in parallel regions (shared, private, reduction) - helping to manage the data scope & lifecycle across threads
Runtime Library Routines → offers a set of library routines for various purposes like setting number of threads, querying number of threads
Environment Variables → control aspects like number of threads to use etc.
pragma omp parallel for → directive that tells compiler that next loop should be executed in parallel
default(shared) private(i) → tells the compilers all variables except i are shared by all threads
schedule(static,chunk) → tells the compiler that iterations of the parallel loop should be distributed in equal sized blocks to each thread (chunk parameter specifies the size of each chunk of iterations) & these chunks of iterations are statically scheduled
reduction(+:result) → performs a reduction on the variables that appear in the argument list (combining values of private copies of result from all threads into a single value)
Static Scheduling: effective when workload is evenly distributed across the iterations of the loop - minimises runtime overhead associated with scheduling - simple way to balance the load when programmer has prior knowledge that each iteration will approximately take the same amount of time
Reduction: simplifies the code by managing the private copies & combining results (otherwise would require manual synchronisation & thread-safe programming) - leads to better performance by minimising synchronisation overhead & allowing concurrent computation - avoids race conditions that can occur when multiple threads try to update a shared variable simultaneously
Static Loop: runs on a single thread in a sequential manner
Shared Memory Parallel Loop (OpenMP): executed by every core - uses self-scheduling to distribute the iterations of the loop across multiple threads in a shared-memory environment
Barrier Synchronisation: synchronisation mechanism which forces all threads to wait until every thread has reached point in the code before any can proceed
Optimisations: working in chunks - instead of fetching one iteration at a time - a thread could fetch a chunk of iterations to work on - reducing overhead of frequent synchronisation for fetching new iteration → avoiding unnecessary barriers: placing barriers only where necessary can minimise the waiting time for threads → exploiting cache affinity: by scheduling iterations that access nearby data elements to the same thread - can take advantage of cache
FetchAndAdd: atomic operation to get the next un-executed loop iteration (atomic operations/lock-based implementation)

Why Multicore: limits of ILP - limits of SIMD parallelism - parallelism at low clock rates very efficient
How to Program a Parallel Machine: explicitly managed threads - parallel loops - message passing

Dynamic Scheduling

Instructions scheduled at **runtime** & processor determines order of execution
Processor inspects state of the pipeline & availability of operands to decide which instructions to execute next (can adapt to runtime events)
Out-of-Order Execution:
Instructions can be issued in order - have their dependencies analysed - & then be executed out-of-order
When Instruction Fetched - may be able to execute instruction or may need to be shelved until operands are available (due to dependency) & FUs available
Lifecycle of Instruction: instruction issued - sometime later begins execution & then completes execution passing operands to instructions that depend on it

Tomasulo:
Algorithm that decouples the dispatch of instructions from availability of their operands & allows OoO Execution
Issue: collects operands from instructions source registers - consults registers & allocates instruction to reservation station
Reservation Station: shelf where an instruction sits when waiting to use a FU & not ready to go get (holds opcode & 2 operands) - both operands must be present before FU can proceed
Registers: value + tag (if tag NULL - value is valid & no current in-flight instruction producing a value for that register) (if tag not NULL - tag is id of FU producing value of that register)
Common Data Bus: dynamically managed bus that forwards result from FUs that produce value to FUs that depend on value - bypassing registers
Registers: tag that connects dependencies (builds dependency graph)
Reorder Buffer (ROB): queue/FIFO that manages the results of OoO Execution to ensure in-order commit (holds dest reg + value/tag)
Commit-Unit: processes instruction in-issue order - taking completed instructions from head of ROB & updating values of commit-side registers
Commit-Side Registers: only externally visible state of the program - maintaining a consistent program state (which can be returned to in case of mispredicted branch/exception)
Speculative Tomasulo Algorithm:
Issue: If Reservation Station & Reorder Buffer Slot Free - Issue Unit takes Source Operands for Instruction & allocates FU
If value for each source register valid then value in register copied to operand in allocated FU & destination register has tag updated with the id of the allocated FU
If source register value not valid - tag copied to operand of FU (rather than value) representing FU that value is dependent on
Execute: monitor CDB for Source Operands (if values not ready) & execute FU when both in RS
Write Result: When Instruction finishes - FU broadcasts result on CDB to all awaiting Registers/RSs/ROB with matching tag indicating that FU will produce result & values updated (results passed directly from dependent FU to Register/RS/ROB)
Commit: commit-side registers updated with ROB result (when an instruction is present at the head of the ROB & its result is present: update the commit-side register with the result & remove instruction from ROB)
Speculative Tomasulo splits machine: speculative-side & commit-side
Hazards & Tomasulo:
Tomasulo solves RAW/WAR/WAR Hazards by dynamically allocating operands to RSs at Issue Time (decoupling operands from RFs)
RAW: once write instruction executes - passed value to Reservation Station of read instruction directly (eliminating RAW Hazard)
WAR: Read Instruction has value itself/tag of FU dependent on - does not matter if the write executes before value fully read as value/tag already in Reservation Station before Issue Time of Write
WAW: second write instruction overwrites the tag in the destination register of the first write instruction (any following instructions reading that value will get tag of FU for second write instruction)
Register Renaming: used to eliminate false data dependencies that arise due to the reuse of architectural registers by successive instructions
Processor can execute more instructions in parallel & exploit ILP
Tomasulo effectively renames destination register to a reservation station
Tomasulo Drawbacks: Complexity: has many associated stores at high speed & multiple comparators monitoring bus in parallel
Performance Limited by CDB: each CDB must go through multiple FUs & no. FUs that can complete per cycle limited to 1
Parallelism: Tomasluo enables multiple FUs to operate in parallel (even across loop iterations) & dynamically adapt to unexpected execution order

Branch Misprediction: when the branch (predicted not taken) reaches the head of the commit queue → could be a misprediction (branch was actually taken) - all issued but not committed instructions (after the branch) are erroneous - trash all of the ROB entries - use commit-side registers to reset values of issue-side registers → reinstating the register state at conditional branch point - correct branch target instructions are fetched & issued (refilling the machine & restarting execution) - reservation stations of uncompleted speculatively-executed instructions cannot be re-used until the functional unit has completed - branch misprediction penalty → takes some time to refill & restart the machine will useful work
→ does not roll-back as soon as the branch is found to be mispredicted (waits for branch instruction to reach the head of the commit queue)
if multiple branch instructions in-flight → on a branch misprediction all ROBs will be trashed → including the any other conditional branch instructions

Store & Loads with Speculation: need to ensure stores are not sent to memory until the store instruction is committed & need to ensure load instructions get the correct data
Solution: if/when the addresses of a load & all preceding uncommitted stores are known: if none of the store addresses match the load → load can proceed / if the address of the load matches the address of an uncommitted store → forward the store's data to the load
→ must add load unit & store buffer - store buffer collects the uncommitted store instructions - when a load instruction → waits in the load unit until its address & all preceding store addresses are known - when load address & all preceding store addresses are known → proceed load instruction if no match, & forward store's data to load if match (alternatively can speculate & add a forwarding predictor to improve speculation & significantly increase performance)

Register Update Units (RUUs): (Alternative to ROBs - a Unified ROB & Reservation Station)
In Tomasulo & ROB → registers & ROB entries have a tag (every register, ROB entry, reservation station needs a comparator to monitor the CDB)
With RUU → the tags are the ROB entry numbers -so the ROB is indexed by the tag on the CDB (ROB thought of as a set of registers) & at instruction issue time → ROB entry allocated for instruction in question → dynamic remapping of registers in instruction set to ROB entries the ROB entries are a renamed destination register
Advantages:
Common Pool of Instructions in the RUU & do not have to commit to which FU we are going to use (in Tomasulo 1 FU per Reservation Station) - RRU defers this decision to dispatch time
Only Issue-Side Registers & RUU have to monitor bus
RUU can be indexed as a RAM - giving considerable efficiency advantage (do not need comparator for every entry)

Pipelining in Dynamic Scheduling:
Deeper Pipelines are often used & each stage is made shorter to allow for higher clock frequencies
More Stages increases the Branch Misprediction Penalty
Dynamic Scheduling can deal better with **hazards** as they can reorder instructions on-the-fly to keep the pipeline full
Deeper Pipelines can perform more speculative work → pays off if speculation correct

Advantages of Dynamic Scheduling: reduced dependence on compile-time instruction scheduling (& compiler knowledge of hardware → compiler can generate code without knowing the particular hardware/processor code will be running on) - handles dynamic stalls due to cache misses (which compiler cannot know about) - register renaming frees architecture from constraints of the instruction set (mapping limited number of registers in instruction set to a much larger set of physical registers in the machine)
Disadvantages of Dynamic Scheduling: increases pipeline depth & therefore misprediction latency - increases power consumption & area (→ more hardware) - increased complexity & risk of design error (large design space → requires more verification & validation - hard to predict performance & hard to optimise count

Shared vs Distributed Memory (2)

Distributed Memory Parallel (MPI - Message Passing Interface):
a standard API for parallel programming using message passing - defines the syntax & semantics of a core of library routines for writing portable message-passing programs on parallel computers - views a parallel program as a collection of processes that have their own local memory - processes communicate & synchronised by sending & receiving messages
Features: point-to-point communication → sending & receiving messages between pairs of processors (with functions like MPISend & MPIRecv) - collective communication → operations involving groups of processes, such as broadcasting a message to all processes or gathering messages from all processes - synchronisation → provides mechanisms for synchronising processes
Single Program Multiple Data (SPMD): parallel programming model - multiple processors executing the same program - operating on different sets of data - each processor must figure out which part of the task needs to perform
SPMD Advantages: highly scalable → each processor operates independently - making SPMD suitable for multicore processors or large distributed memory supercomputers - flexible → allows each processor to execute tasks independently - simplified programming → as each processor executes the same program
OpenMP vs MPI: OpenMP: easy to implement (only need to add pragmas to code & code runs in parallel) - limited scalability (best suited to systems with a limited number of processors sharing memory (i.e. multicore processors) - hides communication (programmer does not have to think about communication or allocate storage) - unintended sharing (can lead to difficult bugs)
MPI: added complexity (programming must define communication & make data passing explicit) - highly scalable (can be used effecitvley on large distributed systmes) - may require more copying of data (need to copy data between memory of different processors) - communication & synchronisation explicit (allows programmer to schedule computation & avoid unnecessary overlaps)

NetBurst

Pentium III: (slower clock rate & less pipeline stages)
To avoid copying register contents around → rather than storing register values in issue-side registers → RAT
Register Alias Table (RAT) keeps track of the latest alias for logical registers → keeping pointers to ROB entries
Issue-Side Register may map to ROB entry (uncommitted), or to a RRF entry (committed) commit-side register file → RRF (retired register file)
If issue-side register does not point to any uncommitted instruction, will point to RRF when instructions are committed → switch pointer from ROB to RRF (where the value has been committed to)
Data is copied from the ROB to the RRF

Pentium 4: (NetBurst) (very high clock rate & has many pipeline stages)
Separate the allocation of physical registers
When instruction is committed → data is not copied from ROB to RRF → stays in RF & Retirement RAT pointer points to location in RF
Committed state of the machine represented by physical registers in RF that are mapped to by logical registers in Retirement RAT
Need to know for each register in RF → which ROB entry corresponds to value which will be assigned to the register
→ as instructions issued, allocate a ROB entry & RF register, RF pointer to ROB entry which will produce its value
ROB only contains status fields
When an instruction complete, RF entry is updated, ROB entry is marked as completed, & commit unit processes ROB entries in sequence & updates Retirement RAT to point to updated retired register
ROB can be managed as a queue (in Pentium 4 → RF can be allocated in dynamic way, pointing Frontend RAT to dynamically allocated register)
Registers are not freed → when an instruction commits, Retirement RAT points to the register → later, when value is overwritten → RF entry can be reallocated

Branch Prediction Alternatives

Objective: attempt to solve control hazards
Amdahl's Law: the performance improvement gained by optimising a single part of a system is limited by the fraction of time that improved part is used (an n-issue processor that has been optimised to issue n instructions per cycle & has a lower CPI will suffer a greater relative impact from control stalls)
Enough Threads per Core: fetching from multiple PCs in a round-robin fashion & each thread as own set of registers (indepdent threads)
Multiplexing multiple threads onto a single core maximises utilisation of processor & provides more time to determine branch outcome (without needed prediction)
Enough Multithreading → eliminates control hazard problem (→ although performance of a single thread dominates the performance & single thread will not perform efficiently)
Predication: avoid branch prediction by turning branches into conditionally executed instructions
Processor is extended with predicate (p) registers (1bit) which hold the outcome of conditional tests & condition evaluated & put into predicate registers
If predicate has value 1 → instructions that depend on that predicate register are executed
Eliminates Control Hazard (avoiding conditional branches) & Converts into Data Hazard
→ Branch Prediction preferred for larger loop bodies - as can jump over the loop body rather than having to run through predicate instructions before discovering predicate is false
→ Predication preferred when issuing many instructions per cycle - as a conditional jump in a sequence of instruction will jump out of that sequence of instructions - leaving that sequence underutilised - predication could pack that sequence of instruction with predicated instructions (potentially useful)
→ Predication preferred if it is difficult to predict branches (avoids frequent large misprediction penalties)
Delayed Branches:
Define a Branch to take place after a following instruction (the following instruction has already been fetched - execute it & then take the branch)
Fill Delay Slot with Useful Work → Instruction from Before Branch/From Target Address(useful when branch taken/From Fall Through(useful when branch not taken)
Usefulness of Branch Delay Slot → Function of Compiler Optimisation
Limitations of Delay Slots: additional complexity (added to Compiler) - increased code size & reduced efficiency (may + NOPs) - forward compatibility issues (if architecture with branch delay slots evolves)

Branch Direction Prediction

Branch Predictor: want to fetch the correct (predicted) next instruction without any stalls - need prediction before preceding instruction (branch) has been decoded
Branch Direction Prediction: predict conditional branch taken or not (takenness)
Branch Prediction Schemes (Takenness): 1-bit BHT/2-bit BHT/Correlating BHT/Tournament Branch Predictor
1-Bit Branch History Table:
Low-Order Bits on PC address index a BHT of 1-bit values - 0/1 (branch not taken/taken last time) (no address check - low-order bits only)
When Branch executed → update the BHT with the corresponding PC index to 0/1 depending on branch not taken/taken
When same Branch encountered again → index the BHT & predicted based upon whether the branch was taken/not taken last time
Aliasing: possible mispredictions if 2 different branch instructions map to the same BHT entry
Loops: in a nested loop - a 1-bit BHT will causes 2 mispredictions (on on first loop pass & one on second loop pass)
2-Bit Branch History Table:
Prediction changes only if misprediction twice (fixing the loop case & works well for loop intensive applications)
4 States: when encountering a branch - updates the state in the BHT for that branch index (incrementing if taken & decrementing if not taken)
Bimodal Predictor: 2-bit BHT Predictor (states) represents a 2-bit Saturating Counter/Bimodal Predictor & exploits the highly biased property of branches (either almost always taken or almost always not taken)
n-Bit BHT:
Increasing BHT Bits: 1-Bit usually worse - 2-bit Often Very Good - 3-bit usually not better
Increasing BHT Capacity: more entries in BHT table - prevent aliasing (any program with many branches should benefit from increased BHT capacity - little performance difference in reality)
Correlated BHT (Global History):
Global History: the taken/not-taken history for all previously executed branches
Correlated BHT: uses m most recently executed branches (implementing an m-bit Branch History Register/BHR)
BHR: a shift register recording taken/not-taken direction of last m branches (when branch encountered - discover takenness & push 0/1 into shift register)
g-select BHT:
Correlated BHT that uses Global Histroy to select from a number of BHTs (according to the behaviour of the last m branches (exploiting correlation between branches)
→ concatenates PC address bits & branch history register bits, to exploit the correlation between successive branches, & determine where prediction will be stored & found
Optimum Choice for Bits of Branch History & PC: 6/6
gselect → many combinations of n & m
global → use only the global histroy to index the BHT - ignore the PC of branch being predicted
gshare → arrange bimodal predictors in a single BHT, but construct its index by XORing (rather than concatenation) low-order PC address bits with global branch history shift register → claimed to reduce conflicts **(optimum choice program dependent)**
Benchmarks: many SPEC benchmarks have less than a dozen branches responsible for 90% of taken branches - & may not represent real programs (do not test branch predictor capacity) (must be careful of appropriateness of benchmarks)
Tournament Predictors:
uses two predictors - one based on global information & one based on local information - combined with a selector (driven to predict the correct predictor to use)
→ delivers better prediction with fewer transistors

Power

Dynamic Power: power consumed when signals change (when a transistor changes start from 0 to 1 or vice versa) - proportional to the square of the supply voltage
Static Power: power consumed when gates powered-up (by leakage current from transistors in a static state)
Dennard Scaling: dynamic power gets smaller if we make the transistors smaller & so can increase clock rate with smaller transistors - end of Dennard Scaling → as transistors become smaller - static power starts to dominate - particularly at high voltages (needed for high clock rates) & so cannot increase clock rates to get performance
Power vs Clock Rate: power increases with clock rate - due to high static leakage due to high voltage - & high dynamic switching
Clock vs Parallelism: much more efficient to use lots of parallel units at a low clock rate & low voltage (less energy per operation)
Solutions: compute fast & then turn system off - compute just fast enough to meet deadline - clock/power gating - turn functional units/cores off when they are not being used - dynamic voltage/clock regulation - reduce clock rate dynamically - reduce supply voltage dynamically - run on lots of cores at a low clock rate - turbo mode

Branch Target Prediction

Branch Target Prediction: predicting target address of branch instruction
Target Prediction → needed for Indirect Branches (e.g. Indirect Branch to Return Address on Stack)
Direct Branch: provides target address in instruction
Indirect Branch: target address not directly specified in branch instruction - instruction specifies a register/memory location containing the target address (e.g. return address, indirect jumps, indirect function calls)

Branch Prediction Schemes (Target): Branch Target Buffer(BTB)/Return Address Predictors(RAP)

Branch Target Buffer: given the current PC - tells you what the next PC should be (while fetching current branch instruction → predicting what next instruction should be)
Table containing Branch PC & Predicted PC
Predicted PC: address to which the branch instruction transferred control to the last time it was taken
Processor predicts that the branch will be taken again → starts fetching instructions of predicted PC
Processor predicted the branch will be not taken → increments the PC normally (PC+4)
BTB in the Fetch Stage of the Processor & checked in parallel with every fetch to predict if fetched instruction will be a taken branch
When a Taken Branch is committed - update the BTB with the Branch's Target Address & tag of branch instruction address
BTB acts as a cache of branch target addresses accessed in parallel with the I-Cache in the Fetch Stage & is only updated by taken branches
BTB Indexed by low-order bits of PC of Fetch & high-order bits of PC used to check tag (hit?)

Branch Misprediction Penalty: in 5-stage pipeline - if misprediction - must override the PC for current fetch & disable the MEM & WB stages for mispredicted instruction (misprediction penalty of one cycle)

Combining BTB with Direction Prediction: simultaneously check the BTB & direction predictor in parallel with I-Cache to determine predicted takenness of branch (if predicted taken & BTB Hit → use target from BTB - otherwise → increment PC normally)
Bigger Slower Predictor can also be used in combination (has much better prediction performance) with a smaller/faster predictor → loses only one cycle if prediction differs - rather than full misprediction penalty
BTB in Dynamically Scheduled Processor: update the BTB when a branch is committed (could update BTB when branch outcome is known - but if discovers a mispredictor the BTB has been misupdated & leads to a further misprediction)

Return Addresses: location directly after a subroutine (JSR) is called
Return Address set to the Predicted PC of the Return Instruction in the BTB (next time returns from function will return to the instruction after the first call - based on the value in the BTB)
Return Addresses form a stack & should be easy to predict
Return Address Predictor (RAP): hardware stack of return addresses
→ when a JSR is decoded - pushes the return address (address immediately after) onto the RAP stack - the RAP stack keeps track of all return addresses & is updated on every JSR/Return Instruction (when JSR decoded RA pushed to stack & when Retrun decode RA popped of the stack)
Popping & Pushing to the RAP done in the decode stage - when mispredictions are detected (so mispredictions do not impacts the RAP stack)
→ when the fetch stage encounters a return instruction - indexes the BTB with the current PC & if the predicted PC is a Return Address - address at the top of the RAP used as predicted PC rather than BTB value
RAP could mispredict a branch target if the call stack deepr than the RAP stack (returns empty) or the RAP prediction is wrong (RAP overwritten/stack pointer changes)
Dynamically Scheduled Processor - BTB updated when branch committed (if wait for this to update RAP - RAP may lack return prediction & if RAP speculatively updated - may be incorrect return address)

Branch Prediciton & Multi-Issue: may encounter 2+ branches in packet of instructions - all the BTB needs to predict is the next instruction to fetch (does not matter which branch is responsible) - bigger slower predictor may later resteer the processor if it has a better prediction that should over-ride the BTB

Multithreading

Instruction Issue: sequential machine - can execute a single instruction per cycle - reduced FU utilisation due to dependencies & corresponding stalls
Superscalar Issue: dynamically scheduled machine - executes multiple instructions per cycle - more performance but lower FU utilisation (lost issue opportunities)
Predicated Issue: improve utilisation of superscalar processor adding predication - less lost issue opportunities (but some results thrown away)
Chip Multiprocessor: multiple processor cores on a chip - execute threads in parallel on separate cores - limited utilisation when only one thread running
Fine Grained Multithreading (FGMT): different threads in different cycles - multiplex multiple threads in parallel on a single core - threads fetched in a round-robin fashion - fills pipelining stalls with other thread instructions (intra-thread dependencies may still limit performance) (if only one thead to run, must wait to be fetched)
Simultaneous Multithreading (SMT): dynamic scheduling of operations from pool of threads (dynamically scheduled machine) - maintains a PC for each thread - fetches instructions from multiple threads per cycle & packs instructions together for simultaneous dispatch in parallel - instructions can be dynamically scheduled from common pool between all threads (if only one thread to run, runs as fast as possible)
SMT gives significant performance improvement (factor of 2 with 4 simultaneous threads)

SMT Pipeline: same as out of order pipeline + dynamic scheduling of instruction fetches of 4 PCs in parallel - instruction cache contains allocations from different threads - issue-side register alias tables (register maps) needed for each thread (containing software-visible registers for that thread) - common RUU/queue - allocates physical registers dynamically to each threads registers from a common pool - dynamically executes independent of specific-thread - allocates into data cache independent of specific-thread - writes to registers & retires independent of specific-thread

Thread Protection Issue: must ensure different threads operating in different protection domains (i.e. different users) can only access their particular memory

Resources: different threads competing for computational resources (as we wanted to increase utilisation) - but also competing for I-Cache & D-Cache - working set of program increased x number of threads (want threads to be dissimilar such that they have less contentions with each other)

SMT Issues: execution time dominated by slowest thread in system - risk of individual threads running slow & impacting execution time - threads contend for resources (resources should be partitioned per thread or shared on-demand) - one thread may monopolise CPU & block progress of other threads - one thread may be able to observe another's traffic (sidechannel issues)

SMT Memory: threads exploit memory-system parallelism (can get a lot of memory accesses in flight & many levels of memory hierarchy can be in used at once)
Latency Hiding: overlapping data accesses with compute (data accesses of some threads overlap with compute of other threads)
No. Threads Limited by No. Registers: threads need lots of registers (number of logical registers x number of threads) - dynamically scheduled processor pays heavy cost for register renaming (register-by-register) → rather than dynamically register renaming (as done in dynamically scheduled processor) - statically partition register file based on the number of registers each thread actually needs - allows more threads to run on machine (maximum occupancy)
Occupancy → maximum number of threads chosen to run concurrently on processor (chosen as a tuning parameter)
Tradeoff: lots of lightweight threads (higher occupancy to maximise latency hiding) vs fewer heavyweight threads (to benefit from many registers)
Static Partitioning: mapping threads into the register file (logical to physical mapping)

Sidechannels

Side-Channels: what can we infer about another thread by observing its effect on the system state? through what channels? how can we trigger exposure of private data? how can we block side-channels?
Spectre v1:
→ defeats bounds checking within a single process's virtual address space - uses branch takenness misprediction to expose a side-channel due to speculative instruction execution - no good mitigation
Meltdown: extends spectre v1 to access any data currently in the process's virtual address space → even if marked supervisor only - exploits a common defect in processor design → where the check for sufficient privilege is performed at commit time - works as OSs would try to avoid having to change the page table when handling a system call or interrupt - removing this optimisation avoids the problem
Spectre v2: uses jump target prediction to choose which code is executed in the address space of another process - by choosing (or more likely finding a way to insert) suitable code → a speculative execution sidechannel can be exploited so the attacker can read the OSs data - this can be mitigated by preventing the attacker from being able to influence the victim's branch target prediction (e.g. retpolines)

Static Scheduling

Instructions are scheduled at **compile time** & compilers tries to optimise order of instructions to minimise hazards
Scheduling is determined **ahead of execution** & cannot adapt to runtime events
Loop Unrolling: optimisation technique used to increase the speed of the loop's execution by decreasing the overhead of loop control & increasing parallelism (increases the number of instructions in the loop body & decreases the number of loop control operations - trading off increased code size for decreased loop overhead)
Software Pipelining: reorganises loops so that each iteration is made from instructions chosen from different iterations of the original loop (static overlapping of loop bodies)
Changing Instruction Set:
Superscalar Processors: decide on the fly how many instructions to issue in each clock cycle (capable of issuing multiple instructions per cycle)
→ have to check for dependencies between all n pairs of instruction in a potential parallel instruction issue packet - hardware complexity of figuring out the number of instructions to issue is $O(n^2)$ - leads to multiple pipeline stages between fetch & issue
Solution: allow compiler to schedule instruction level parallelism explicitly & format the instructions into a potential issue packet (& hardware does not need to check explicitly for dependencies)

VLIW: each instruction has explicit coding for multiple operations
→ all operations the compiler puts into long instruction word independent → can be issued & executed in parallel
→ need compiling technique that schedules across several branches (simplifies hardware & complicates compiler)
EPIC (IA64): ISA that exposes parallelism to compiler - issues instructions in instruction groups
Instruction Group: a sequence of consecutive instructions with no register data dependencies → all instructions within a group could be executed in parallel - if sufficient hardware resources exist & if any dependencies through memory are preserved → group can be arbitrarily long (compiler must explicitly indicate boundary between one instruction group & another) → compiler places a stop between two instructions that belong to different groups → compiler telling hardware explicitly where parallel issue starts & stops
Hardware Support for Exposing More Parallelism to Compiler:
Register Stack: general purpose registers are configured to help accelerate procedure calls using a register stack
→ maintaining a logical mapping between logic registers included in instructions & physical registers in machine → explicitly changing logical mapping via a function call
→ a special register called the current frame pointer (CFM) points to the set of registers used by a given procedure
Predication:
conditions can be computed & stashed in a predicate register → used to control whether and instruction executed → almost all instructions can be predicated
→ compiler can move instructions across conditional branches → can pack parallel issue groups → may also eliminate some conditional branches completely → avoids branch prediction & misprediction when a branch would break a parallel issue packet → move instructions & predicate them

Vectors & SIMD

Vector: sequence of data elements of the same type that are processed by a single instruction
Vectors require Vector Instruction Sets - use Automatic Vectorisation (compiler selects & used vector instructions) - can use lane-wise predication (vectorising conditionals)
SIMD (Single Instruction Multiple Data): execution of a single instruction applied to multiple data elements simultaneously in parallel
In SIMD - each processor unit that performs the operation works on a different data elements - form a vectorisation in which a single operation is applied to a vector of data- uses registers & instructions that hold/execute whole vectors of operands at once
Advantages of SIMD:
Reduces Turing Tax (overhead of general-purpose compute machine) by amortising fetch-execute overhead (many data elements executed on a single fetch)
Increases ILP (reducing instruction count & improving data throughput)
Significantly speeds up computational tasks that are parallelism
Arithmetic Intensity: ratio of computational operations to memory operations (Number of FLOPs / Number of Bytes Transferred)
High Arithmetic Intensity → performs many operations per memory access (typically compute-bound & peak performance GFLOP/s limits performance)
Low Arithmetic Intensity → typically memory-bound & memory bandwidth peak GB/s limits performance
Roofline Model: arithmetic intensity vs achieved compute performance → predicting the maximum possible performance of a program (given machines peak performance & memory bandwidth)
If Memory BW limits performance: optimise data movement - organisd data accesses to reach memory bandwidth - reduced amout of adata program needs to move & increase arithmetic intensity → performance reaches a higher factor of peak performance
Vector Instruction Set Extensions (AVX512): extends the scalar processor with vector registers (each 512bits wide) - extended register used to store 8 doubles/16 floats/32 shorts/64 bytes (vector instructions for all operands) - instructions executed in parallel in 64/32/16/8 lanes - contains predicate registers - each predicate register holds a predicate per operand (per lane) - k predicate registers hold up to 64bits (for maximum of 64 lanes)
Lane: element-wide slice of a sequence of vectors
Predication: only performs operation on lanes with corresponding predicate bit set activate
Automatic Vectorisation: compiler converts scalar operations into vector operations (compiler automatically vectorises the scalar operations)
→ automatic vectorisation performed by compiler during the code compilation process → compiler analyses the code - identifies loops/sequences of operations that can be vectorised - & generates corresponding vector instructions → automatic vectorisation commonly done in loops → if the iterations of a loop are independent & involve operations on arrays/sequences of data → compiler can replace scalar operations with vector operations handling multiple data elements of the array in parallel
Conditions in which Compiler can Vectorise code: known loop bounds → compiler must know how many times a loop will execute (knows how many loop iterations need to run in parallel) - no dependencies → each iteration of loop must be independent of others (iterations cannot run in parallel if dependencies between iterations) - alignment → data should be aligned in memory for vector operations
Problems: loops have complex control structure (e.g. may have unknown loop bounds) → data dependencies often exist which are hard for compiler to understand → issues with memory alignment
Solutions: restructure code to simplify loops (so has known bounds, no dependencies) - using compiler hints (such as #pragma ivdep) to inform compiler about independence of loop iterations (so compiler does not need to infer this itself) - explicitly aligning data structures in memory - using SIMD Intrinsics → function-like constructs that directly utilise SIMD instructions - giving programmer more control over vectorisation - OpenMP Pragmas → directives that can be used to instruct the compiler to vectorise certain loops/code sequences
If compiler cannot vectorise code: **SIMT** → a single instruction stream (vector) is executed across multiple threads - each thread handles different data (a different lane) - uses prediction to allow different threads to follow different branches while remaining executing in parallel
Vector Pipelining: vector instructions are executed serially (element-by-element) using several pipelined functional units
Vector Chaining: vector instructions are executed serially (element-by-element) using several pipelined functional units - functional units form a long pipelined chain - (without vector chaining each vector instruction would need to load & store data to memory from which the next instruction would retrieve this data → chaining passes the data between instructions rather than accessing memory)
SIMD Architectures: reduce Turing Tax (more work with fewer instructions & less overhead - relies on programmer/compiler) - more complicated loops cause issues & lane-by-lane predication allows conditionals to be vectorised but branch divergence may lead to poor utilisation
Vector ISA: provides a broad spectrum & raises level of abstraction giving more choices to programmer/compiler

Synchronisation & Memory Models

Synchronisation: coordinate of concurrent processes to ensure correct execution
In a Multi-Process System - different processe may run in parallel & access/modify shared resources - & without synchronisation leads to race conditions where outcome dependes on order of execution
Race Condition: situation in which behaviour of system depends on the execution of multiple processes (accessing shared resources) running in parallel (which process executed first may vary & can affect correctness)
Lock/Mutex: used to ensure that only one thread can access a resource at a time (when a thread acquires a lock - no other thread can access the locked section until the lock is released)
Semaphore: a generalised lock - maintains a count & a thread can decrease or increased the count - used to control access to a resource pool/synchronisation activities
Barrier Synchronisation: used to make sure multiple threads reach a certain point in the execution before any of them can proceed
Atomic Operations: operations that is performed as a single step that cannot be interrupted/observed in an incomplete state
Synchronisation Bottlenecks:
Fast Non-Contended Path (where there is no competition for resources) → synchronisation should still perform efficiently
Efficiency in High Contention → when many processes are competing for the same resource - synchronisation should still perform efficiently
Fairness → all processes should have equal opportunity to access shared resources - preventing scenarios where some threads are starved of resources while others monopolise them
Atomic Operations: (Single Atomic Load & Store)
Test-and-Set: atomic operations that tests a value (typically a synchronisation variable such as a lock) & sets it if a certain condition is met (i.e. lock is free) - used to implement mutexes where a thread can acquire a lock before entering a critical section
Fetch-and-Increment: atomic operation that reads the current value of a memory location & then increments a value atomically
Atomic Exchange: atomically swaps a value between a register & memory location - used to implement synchronisation mechanisms (i.e. thread acquiring a lock would use atomic exchange to swap 1 into memory location of lock variable)

GPUs

Workloads Consisting of Thousands of Threads: never speculate (always another thread ready to execute) - no speculative branch execution (perhaps no branch prediction) - can use multithreading (FGMT or SMT) due to high parallelism to hide cache access latency (& perhaps memory latency)
Control Overhead: launching 10,000 threads at once - taking branches in different directions - accessing random memory blocks
GPUs vs CPUs: simpler cores - much greater area devoted to computation rather than control logic - many functional units (implementing the SIMD model) - much less cache per core (as thousands of thread with super-fast context switching that can use fine-grained multi-threading to hide cache access latency) - dropped sophisticated branch prediction mechanisms
Thread: smallest unit of execution on a GPU (like a lane on a SIMD CPU)
Warp: specific collection of threads that are executed simultaneously by GPU (like threads in CPU)

GPU Microarchitecture: many fetch-execute devices (16 SMs/Cores) - each core uses fine-grain multithreading FGMT to run 32 warps per SM - high number of warps to tolerate high memory access latency - wraps run on SMs - threads run on SPs - MT Issue selects which warp to issue from in each cycle (FGMT) (which warp has received data from memory & is ready) - each warp's instructions are 32-wide SIMD instructions (each warp contains 32 threads) - instruction executed in four steps (using 8 SPs) - each thread in a warp executes the same instruction in lock-step on different data - SIMT (single instruction multiple thread)→ each thread executes in a SIMD fashion + has its own instruction address & register state (as divergence can occur if threads in a warp need to execute different instructions due to threads having to cover all branches of a conditional - leading to inefficiences) - workload for each thread should be designed to minimise divergence & maximise throughput (not having to execute different branch paths) - uses lanewise predication (to avoid divergence)

Memory: each SM has local explicitly-programmed shared scratchpad memory (user-managed cache) - different wraps on the same SM can share data in shared memory - SMs have local L1 Data Cache (but no cache-coherency protocol) - machine is programmed by launching 1000s of threads at once (grouped into warps) & flushing L1 Data Cache (data is not exchanged between SMs during execution of an individual kernel & therefore no coherency protocol needed) - multiple DRAM channels on-chip - each including an L2 Cache (each data value can only be in one L2 location - no cache coherency issue)

CUDA (Compute Unified Device Architecture): using GPUs for general-purpose computation - needed to manage thousands of threads
Serial CPU Code & parallel GPU code (Kernel)

GPU Kernel: each kernel is a C function (which can be run by 1000s of threads) - each thread executes an instance of a kernel function on a single element of data - a group of threads form a thread block (larger structure containing several warps) - blocks are unit of allocation into SMs - thread blocks are organised into a grid - threads within the same thread block can synchronise execution & share access to local scratchpad memory
Hierarchy of Parallelism: hierarchy of parallelism to handle thousands of threads - thread blocks are dynamically allocated to SMs & run to completion - threads within a block run on the same SM & can share data & synchronise - different blocks in a grid cannot interact with each other

SIMD (Single Instruction Multiple Data) → single instruction operating on multiple data elements simultaneously (same operation applied across different data elements in parallel) - SMT means a small number of threads run on the same core to hide memory latency - divergence not possible - exploits spatial locality through adjacent threads & adjacent loop iterations

SIMT (Single Instruction Multiple Thread) → multiple threads execute the same instruction at the same time on different data → like SIMD - but each thread in SIMT can follow its own control path (while all threads start executing the same instruction - they can diverge based on their specific data & control conditions) → allows more flexibility than traditional SIMD & allows multiple warps running on the same core to hide memory latency
→ processor manages & executes many threads in parallel (each thread considered independent) - grouped into warps → one thread per lane → adjacent threads (warps) execute in lockstep → SMT means multiple warps running on the same core to hide memory latency → each thread can follow own control path - divergence possible → exploits spatial locality through adjacent threads accessing adjacent data → load can result in different addresses accessed by each lane

SM:
SM has a SIMT multithreaded instruction unit that creates, manages, schedules, & executes threads in groups of warps → each SM manages a pool of warps (FGMT ways) → individual threads composing a SIMT warp start together at the same program address (free to branch & execute independently) → at instruction issue time - select ready-to-run warp & issue next instruction to that warp's active threads - each instruction operates on a 32-wide vector → each thread (element-wide slice of a vectors) contains the state of a CUDA thread → when successive instructions from the same warp are executed - progresses the execution of all threads in the warp → warps interleaved with other warps through FGMT (e.g. to hide access latency of warp accessing memory) → the FGMT arrangement means that while one warp is executing - other warps are scheduled/executed → demonstrating how the GPU maximises throughput by keeping as many threads active at any given time → SIMT shares SM Instruction Fetch & Issue unit across 32threads - but requires full warp of active threads for fully performance efficiency

Branch Divergence:
Threads can diverge in a Warp (a warp serially executes each path - disabling some of the threads & when all paths complete the threads re-converge) - Divergence only occurs in a Wrap (different warps execute independently)
To optimise program: minimise divergence within threads in a warp (good utilisation)
→ Predicate Bits can be used to handle divergent paths without branching

Coalescing:
→ ability of threads in a warp to access contiguous memory locations - allowing them to be combined into a single memory transaction
→ in SIMT - threads with adjacent IDs access data in different lanes (non-contiguous memory locations) & memory accesses cannot be coalesced

Spatial Control Locality:
SIMD:
→ branch predictability → each individual branch is mostly taken or not taken (or well predicted by global history)
SIMT:

→ branch coherence → adjacent threads in a warp all usually branch the same way (spatial locality for branches across threads) → if branches are coherent - good spatial locality → if branches not coherent - many wasted instruction opportunities

Cache Miss Rate Reduction

Reducing Miss Rate in Hardware: change block size/change associativity/change compiler
Larger Block: initially the miss rate improves with increasing block size → due to spatial locality (more data copied into cache at once)
Increasing Associativity: improves (decreases) miss rate → but, cache hit time is increased slightly (due to greater selector logic depth)
Victim Cache: combines fast hit time of direct mapped & miss rate of associative cache (avoiding conflict misses) - add a buffer to place data discarded from cache - direct mapped cache above & fully-associative cache below (buffer/victim cache) - access both in parallel - fully associative can be very small → small energy cost & fast cycle time - direct mapped cache → fast cycle time
Skewed Associative Cache: skewed 2-wat set associative cache → depending on the hash function → all 3 could map to the same set in the left way & all 3 could map to different sets in the right way - with skewed-associativity → can get same miss rate with reduced associativity - harder to write a program that is free of associativity conflicts (cannot predict hash function)
Prefetching: a better way of exploiting spatial locality (fetching data ahead of time before they are needed for execution) + stream buffer → hardware prefetching mechanism - stream buffer exploits spatial locality by prefetching next cache lines before they are needed during execution (predicting that the next cache lines are going to be needed by execution)
Decoupled Access Architecture: much faster to access the cache lines from stream buffer, rather than accessing main memory - decouple integer & floating point operations to allow integer operations to runahead & exploit parallelism to run as fast as possible (with no dependencies to vector operations)
Reducing Miss Rate in Software: prefetch instructions - transforming storage layout - transforming iteration space - loop interchanging - loop fusion
Software Prefetching: reduce miss rate anticipating data needs of a program prefetching that data before execution time with explicit instructions inserted by compiler - effectiveness of software prefetching depends on the timing of prefetch instructions (too close & instruction will stall / too far & prefetched line may have be evicted before used) - care needed to ensure pregetch access does not have unwanted side effects - executing prefetch instructions has a cost (instruction must be decoded & uses some execution resources) - prefetch instruction from main memory only needs small improvement in cache miss rate to be useful due to high memory access latency - commonly hardware prefetching is good enough
Storage Layout Transformations: change way in which variables are declared & address at which data is stored (improves spatial locality & cannot change temporal locality → cannot change order of execution)
Merging Arrays: improves spatial locality by merging two arrays into a single array of compound elements
Permuting a Multidimensional Array: improve spatial locality by matching array layout to traversal order
Iteration Space Transformations: change the ordering in which loops are executed (improves temporal locality → transforms a particular loop & can change spatial locality)
Loop Interchange: change nesting of loops to access data in order stored in memory
Loop Fusion: - Loop Fusion → combine two independent loops that have the same looping & some variables overlapping
Blocking: improve temporal locality by accessing blocks of data repeatedly (rather than going down whole columns or rows)
Reduce Hit Time: use a really small L1 cache - use a pipelined cache (improves BW) - use a multi-bank cache (improves BW) - use a direct-mapped cache - pass data forward while checking tags in parallel - use way prediction - take address trnaslation of the critical path (access the TLB in parallel with L1 Cache)