Parallelism

1.1 Instruction-Level Parallelism (ILP)

Pipelining serves as the fundamental technique for ILP by overlapping the execution of multiple instructions. To further maximize ILP, systems can utilize deeper pipelines, which decompose instructions into more numerous, smaller stages. This reduction in work per stage allows for a higher clock frequency (shorter clock cycles). However, as pipeline depth increases, the overhead of hazards and the complexity of hardware logic also scale.

Multiple Issue. To push performance beyond the theoretical limit of a single-issue pipeline, where the Cycles Per Instruction (CPI) is at best 1, architectures employ Multiple Issue techniques. In these systems, the Instructions Per Cycle (IPC) becomes the primary metric of throughput.

- *Static Multiple Issue*: In a static framework, the responsibility of managing parallelism shifts to the compiler. The compiler groups instructions into issue packets, which are perceived by the hardware as a single "very long instruction" (VLIW).
  - *Packet Construction*: The compiler packages instructions based on available pipeline resources (issue slots).
  - *Hazard Management*: The compiler is responsible for detecting and resolving hazards. It must ensure that no dependencies exist within a single packet and manage dependencies between successive packets according to the specific ISA rules.
  - *Resource Alignment*: If the compiler cannot find enough independent instructions to fill a packet, it must pad the remaining slots with `nop` (no-operation) instructions.

  A common implementation is the MIPS Static Dual-Issue processor, which fetches and executes a "packet" of two instructions every clock cycle. To simplify the hardware decoding logic, these issue packets are strictly formatted. They must be 64-bit aligned, containing two 32-bit instructions in a specific order:

  (1) *Slot 1*: An ALU or Branch instruction.

  (2) *Slot 2*: A Load or Store instruction.

  If the compiler cannot find a pair of instructions that meet these criteria and lack dependencies, it must pad the unused slot with a `nop` . This ensures the hardware always receives a predictable instruction pair.

  While dual-issue increases potential throughput, it introduces more restrictive timing constraints for data hazards, requiring more aggressive compiler scheduling:

  - *EX Data Hazards*: In a single-issue pipeline, forwarding allows an ALU result to be used immediately by next instruction. However, in a dual-issue packet, an ALU result produced in Slot 1 cannot be used by a Load/Store in Slot 2 of the same packet. The instructions are executing in the same stage simultaneously, so the result has not yet been generated when the second instruction requires its operands.

- *Load-Use Hazards*: The "use latency" for a load instruction remains one cycle, but the impact is doubled. Because the processor issues two instructions per cycle, a one-cycle delay now stalls two potential instruction slots instead of one.

To maintain efficiency, the compiler must look further ahead in the code to find independent instructions to fill these gaps, a process known as Static Instruction Scheduling.

- *Dynamic Multiple Issue*: Superscalar processors represent a shift in responsibility from the compiler to the hardware. In this architecture, the CPU dynamically decides whether to issue zero, one, or multiple instructions during each clock cycle. The primary objective is to maximize throughput by avoiding structural and data hazards in real-time. While a compiler can still assist by reordering instructions to be "hardware-friendly", it is no longer strictly required for functional correctness. The CPU hardware ensures that the final code semantics remain identical to a sequential execution, regardless of how the instructions were shuffled internally.

  To mitigate stalls, superscalar units often employ Dynamic Pipeline Scheduling. This allows the CPU to execute instructions out-of-order as soon as their operands are available. However, to maintain architectural integrity, the system must commit results to registers in-order. This ensures that if an interrupt or exception occurs, the machine state is inconsistent and predictable.

Loop unrolling is a code transformation technique that reduces loop overhead by replicating the loop body multiple times and decreasing the total number of iterations. It is a key method for exposing Instruction-Level Parallelism in both static and dynamic multiple-issue systems. By executing multiple iterations of the original loop within a single unrolled iteration, the processor significantly reduces the frequency of administrative instructions, which means fewer increments of the loop counter and fewer conditional branches are executed. In deep pipelines, every branch is a potential stall or misprediction risk; unrolling reduces the total number of branches, keeping the pipeline streaming longer.

Speculation. Speculation is a sophisticated technique used in both static and dynamic systems to preemptively execute instructions before the processor is certain they are required. This is particularly effective for overcoming the bottlenecks of branch delays and long-latency load operations. The process involves a "guess-check-recover" cycle: the system guesses the outcome of a control or data flow decision, executes the operation based on that guess, and then verifies the result. If the guess was correct, the operation is committed; if incorrect, the system performs a roll-back to restore the previous architectural state.

- *Compiler Speculation*: The compiler may reorder instructions, such as moving a `load` operation before a preceding `branch`. To handle potential errors (like a memory protection fault on a speculative load), the compiler integrates "fix-up" code to recover gracefully.
- *Hardware Speculation*: The processor uses look-ahead buffers to execute instructions out-of-order. The results are stored in temporary buffers (such as reorder buffers) and are only written to the permanent architectural state once the speculation is confirmed as correct. If the speculation fails, these buffers are simply flushed.

Conclusions. While compilers are powerful, they are limited by the information available at "compile-time". Hardware scheduling is superior for several reasons:

- *Unpredictable Stalls*: Many stalls, such as cache misses, are non-deterministic. A compiler cannot know exactly when a data request will miss the cache, but the CPU can detect the miss and immediately switch to other independent instructions.
- *Dynamic Branch Outcomes*: The specific path a program takes often depends on runtime data. Hardware can use dynamic branch predictors to speculatively execute paths that a static compiler cannot definitively identify.
- *Microarchitectural Portability*: Different implementations of the same Instruction Set Architecture (ISA) may have varying pipeline depths, latencies, and functional units. Hardware scheduling allows the same binary code to run efficiently across different CPU generations without needing to be recompiled for every specific hardware hazard.

Despite the sophistication of multiple-issue and speculation, several "walls" prevent us from achieving infinite parallelism. Programs possess inherent dependencies that create a ceiling for Instruction-Level Parallelism.

- *Data and Resource Dependencies*: Real dependencies, such as a calculation requiring the result of the immediately preceding one, create chains that cannot be parallelized.
- *Pointer Aliasing*: This is a significant hurdle in languages like C. If the compiler of hardware cannot be certain whether two different pointers refer to the same memory address, it must assume a dependency exists, which prevents reordering load/store operations.
- *Limited Windows Size*: A CPU can only "look ahead" at a certain number of instructions to find parallelism. Increasing this window requires massive amounts of power and silicon area, leading to diminishing returns.
- *Memory Bottlenecks*: Modern CPUs are significantly faster than the memory systems that feed them. Limited memory bandwidth and high latency make it difficult to keep deep, wide pipelines consistently full of useful instructions.

Speculation remains the most effective tool for breaking through these limits, but it must be executed with high accuracy. If speculation fails too often, the energy and cycles spent on "wrong-path" execution actually decrease performance below that of a simpler, non-speculative processor.

1.2 Single Instruction, Multiple Data (SIMD)

SIMD (Single Instruction, Multiple Data is an architecture category designed to exploit Data-Level Parallelism (DLP)). By applying a single operation to an entire vector of data simultaneously, SIMD significantly accelerates computationally intensive tasks like digital signal processing (DSP), audio/video encoding, and deep learning inference.

To understand SIMD, it is helpful to contrast it with the traditional processing model:

- *SISD*: This is the conventional scalar processing model (e.g., standard MIPS or ARM code). The CPU executes one instruction to process exactly one data item at a time. To add two arrays of 8 elements, a SISD processor must execute 8 separate addition instructions in a loop.
- *SIMD*: Also known as Vector Parallelism, this model allows one instruction to operate on a "pack" of data elements (a vector). In the same example of adding arrays, a SIMD processor

with an 8-lane vector width can perform all 8 additions in a single clock cycle using one instruction.

ARM NEON: Advanced SIMD. NEON is the ARM implementation of advanced SIMD technology. It functions as a dedicated accelerator integrated into the processor, specifically designed to handle the high-throughput requirements of modern multimedia and AI workloads. There are three primary ways to utilize NEON in an embedded system:

(1) *Auto-vectorization*: The compiler automatically analyzes standard C/C++ loops and converts them into NEON instructions, which is an operation that requires specific compiler flags (e.g., `-O3 -mfpu=neon`).

(2) *NEON Intrinsics*: These are special C/C++ function calls that map directly to NEON instructions. They provide low-level control over the hardware while allowing the compiler to handle register allocation and instruction scheduling.

(3) *Hand-coded Assembly*: For maximum performance, developers can write raw NEON assembly. This is typically reserved for critical "hot spots" in a program where the compiler's output is not sufficiently optimized.

NEON utilizes a dedicated register file that is distinct from the general-purpose ARM registers (`r0 - r15`). This file is 256 bytes large and offers a dual-view architecture, allowing the hardware to treat the same physical storage in two different ways:

- *D Registers (Double-word)*: 32-bit registers, each 64 bits wide (`D0 - D31`).
- *Q Registers (Quad-word)*: 16-bit registers, each 128 bits wide (`Q0 - Q15`).

The registers are aliased, meaning `Q0` is physically composed of `D0` and `D1` , `Q1` is composed of `D2` and `D3` , and so on. This flexibility allows the programmer to choose the vector length that best fits the data type and algorithm. Each NEON register is viewed as a vector of elements of the same type, known as lanes. The number of lanes depends on the data size:

- A 128-bit Q register can hold:
  - 16 elements of 8-bit data (e.g., pixels).
  - 8 elements of 16-bit data (e.g., audio samples).
  - 4 elements of 32-bit data (e.g., floating-point values).
  - 2 elements of 64-bit data.

On many ARM architectures, the NEON register file is shared with the VPF (Vector Floating Point) unit. While they share registers, they serve different purposes: the VFP is a fully IEEE-754 compliant floating-point coprocessor for high-precision scalar math, whereas NEON is a high-speed, parallel engine for throughput-oriented vector math.

1.3 Parallel Processors

The fundamental objective of parallel processing is the interconnection of multiple processing elements to achieve enhanced computational performance. This architecture is generally categorized into three operational scales: task-level parallelism, which prioritizes high throughput for

independent workloads; parallel processing programs, where a single application is distributed across multiple processors; and multicore microprocessors, which integrate several processing units, or cores, onto a single physical substrate.

Power Dynamics and Efficiency. Power efficiency remains a primary constraint in the design of advanced embedded systems. The dynamic power consumption of a circuit is governed by the switching power equation:

$$P_{\text{switch}} = \alpha \cdot C \cdot V_{\text{dd}}^2 \cdot f \tag{1}$$

In this context, α represents the activity factor, C the capacitance, V_{dd} the supply voltage, and f the operating frequency. To optimize for power, designers often reduce the number of pipeline stages, which subsequently lowers the total flip-flop count. Furthermore, the synthesis process itself impacts the energy profile; enforcing higher frequency constraints during logic synthesis typically results in more power-hungry netlists. Multicore architectures mitigate some of these issues by allowing for core specialization, where specific cores are optimized for distinct tasks rather than relying on a single, high-frequency general-purpose unit.

The primary bottleneck in parallel computing is the complexity of the software rather than the hardware. To justify the transition from a uniprocessor to a multicore system, the software must demonstrate significant performance improvements; otherwise, the simplicity of a faster uniprocessor remains preferable. These challenges are primarily rooted in three areas:

(1) *Partitioning*: Dividing the problem into discrete, parallelizable tasks.

(2) *Coordination*: Managing the synchronization between concurrent threads.

(3) *Communication Overhead*: The latency and bandwidth costs associated with data exchange between cores.

Amdahl's Law. Amdahl's Law provides a theoretical framework for predicting the maximum speedup of a system when only a portion of it is improved or parallelized. It establishes that the overall performance gain is strictly limited by the sequential fraction of the program, which cannot benefit from additional processors. The fundamental relationship for speedup is defined by the fraction of the code that is parallelizable (P). In an ideal scenario with infinite processors, the maximum speedup is constrained by the remaining serial fraction ($1 - P$):

$$\text{Speedup}_{\text{max}} = \frac{1}{1 - P} \tag{2}$$

When considering a finite number of processors (N), the execution time is the sum of the time spent on the serial portion (S) and the time spent on the parallel portion (P) distributed across those processors. This is modeled by the equation:

$$\text{Speedup} = \frac{1}{S + \frac{P}{N}} \tag{3}$$

In this mode, S represents the serial fraction, P is the parallel fraction (where $S + P = 1$), and N is the number of processing elements. As N increases, the term $\frac{P}{N}$ diminishes, leaving the serial fraction S as the ultimate bottleneck.

The effectiveness of parallelization is often evaluated through two distinct lenses:

- *Strong Scaling*: This approach keeps the total problem size fixes while increasing the number of processors. The primary goal is to reduce the "time to solution". Strong scaling is heavily governed by Amdahl's Law, as the serial portion becomes increasingly dominant as the parallel work per processor shrinks.
- *Weak Scaling*: In this perspective, the problem size grows proportionally with the number of processors, maintaining a constant workload per processor. The objective is to solve larger, more complex problems in the same amount of time rather than solving a small problem faster. Weak scaling is often modeled by Gustafson's Law, which suggests that parallel performance can scale more linearly if the workload is allowed to expand.

Multithreading and Shared Memory Architectures

Multithreading is a technique that enables the parallel execution of multiple threads by replicating hardware resources, such as registers and Program Counters (PCs), to allow for rapid context switching. Unlike traditional process switching, which involves heavy OS overhead, hardware multithreading minimizes latency by maintaining thread states in the processor pipeline. The efficiency of multithreading depends on the frequency and conditions under which the processor switched execution contexts:

- *Fine-grained Multithreading*: The processor switches between threads after every clock cycle, interleaving instructions in a round-robin fashion. If one thread stalls (e.g., due to a dependency), the pipeline remains utilized by executing instructions from other threads. While this maximizes throughput, it can slow down the execution of an individual prime thread.
- *Coarse-grained Multithreading*: Context switching only occurs during significant stalls, such as an L2 cache miss. This approach simplifies hardware design as it does not require cycle-by-cycle switching; however, it is less effective at masking short-term stalls like data hazards or functional unit contention.
- *Simultaneous Multithreading*: Employed in multiple-issue, dynamically scheduled processors, SMT allows instructions from different threads to execute concurrently within the same clock cycle. By leveraging the superscalar architecture, SMT utilizes functional units that would otherwise remain idle. Within individual threads, dependencies are managed through dynamic scheduling and register renaming.

In an SMP system, the hardware provides a single, unified physical address space accessible by all processors. Communication between threads is managed by synchronizing shared variables, typically through the use of software locks. The performance of these systems is often categorized by the memory latenxy:

- *Uniform Memory Access (UMA)*: All processors share the same memory latency when accessing any part of the physical memory.

- *Non-Uniform Memory Access (NUMA)*: Memory is physically distributed; a processor can access its local memory faster than remote memory blocks assigned to other processors.

When two or more processors share a memory region, hardware-level support for atomic operations is required to prevent race conditions. An atomic read-modify-write operation ensures that no other processor can access or modify a specific memory location between the initial read and the final write. This atomicity is typically implemented via:

- A single atomic instruction (e.g., Test-and-Set).
- An atomic pair of instructions (e.g., Load-Linked / Store-Conditional).
- Hardware-level mutexes or semaphores managed by the memory controller.

Accelerator Control and Execution Models. Effective accelerator integration requires managing the flow of control between the CPU and specialized hardware. This is primarily handled through two execution models: single-threaded (blocking) and multithreaded (non-blocking) control. In a single-threaded environment, the CPU initiates an accelerator task and enters a stall or wait state until the accelerator completes its operation. This sequential execution creates significant idle time for the CPU. Conversely, a multithreaded model employs a split-join architecture. After the CPU triggers the accelerator (the "split"), it continues executing independent tasks while the accelerator runs in parallel. The execution flows then "join" once both the CPU and accelerator have reached their respective synchronization points.

Partitioning and Data Distribution. Partitioning is the process of decomposing a problem into smaller, parallel tasks. This decomposition generally follows two methodologies:

- *Domain Decomposition*: The problem data set is divided into discrete chunks, and each task performs similar operations on different pieces of data.
- *Functional Decomposition*: The problem instruction set is divided. In this model, different tasks perform different operations, often arranged in a pipeline where data flows from one task to the next.

For effective hardware mapping, data is distributed using 1D or 2D patterns. Common distribution strategies include Block, where contiguous data chunks are assigned to a processor, and Cyclic, where data elements are interleaved across processors to ensure a better load balance.

Scheduling, Mapping and Benchmarking. Scheduling involves the assignment of tasks to specific processors while satisfying several constraints. To optimize execution time, designers must account for the task graph, which defines the order of execution, and inter-task dependencies. Furthermore, communication-related overhead, such as the time data spends on the bus or network, must be factored into the total execution cost. For example, a task transfer might consume specific time units on the system bus, delaying the start of dependent tasks on remote processors.

To evaluate these systems, several standardized benchmark suites are utilized:

- *Linpack*: Measures matrix linear algebra performance.
- *SPECrate*: Measures the throughput of parallel runs of SPEC CPU programs.
- *SPLASH/PARSEC*: Suites focused on shared-memory multithreaded applications.

- *NAS*: Kernels focused on computational fluid dynamics.

The Roofline Model

The Roofline model is a visual framework used to determine the attainable performance of a kernel on a specific architecture. It relates floating-point performance to Arithmetic Intensity, which is defined as the ratio of floating-point operations (FLOPs) to the total bytes of memory accessed:

$$\text{Arithmetic Intensity} = \frac{\text{FLOPs}}{\text{Bytes}} \tag{4}$$

The attainable performance is constrained by two hardware limits: the Peak Floating-Point Performance and the Peak Memory Bandwidth. The formula for attainable GFLOPs/sec is:

$$\text{Att. GFLOPs/sec} = \min(\text{Peak Memory BW} \times \text{Arithmetic Intensity}, \text{Peak FP Performance}) \tag{5}$$

Optimization Regions. A kernel's position on the Roofline diagram dictates the necessary optimization strategy:

(1) *Memory-Bandwidth Limited*: When a kernel has low arithmetic intensity, performance is limited by how fast data can be moved from memory. Improvements here require optimizing memory usage, such as using software prefetching or ensuring memory affinity to avoid non-local accesses.

(2) *Computation Limited*: When arithmetic intensity is high, the kernel is limited by the processor's raw speed. Performance can be improved by balancing adds and multiplies, increasing instruction-level parallelism, or utilizing SIMD instructions.

Arithmetic intensity is not always a fixed value; it can scale with the problem size or be effectively increased through caching, which reduces the number of required memory access.