FUNDAMENTS OF COMPUTER ARCHITECTURE

Sunanda Das
C R Manjunath

Fundaments of Computer Architecture

Fundaments of Computer Architecture

Sunanda Das
C R Manjunath

BOOKS ARCADE
KRISHNA NAGAR, DELHI

Fundaments of Computer Architecture

Sunanda Das
C R Manjunath

© RESERVED

This book contains information obtained from highly regarded resources. Copyright for individual articles remains with the authors as indicated. A wide variety of references are listed. Reasonable efforts have been made to publish reliable data and information, but the author and the publisher cannot assume responsibility for the validity of all materials or for the consequences of their use.

No part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereinafter invented, including photocopying, microfilming and recording, or any information storage or retrieval system, without permission from the publishers.

For permission to photocopy or use material electronically from this work please access booksarcade.co.in

BOOKS ARCADE

Regd. Office:
F-10/24, East Krishna Nagar, Near Vijay Chowk, Delhi-110051
Ph. No: +91-11-79669196, +91-9899073222
E-mail: info@booksarcade.co.in, booksarcade.pub@gmail.com
Website: www.booksarcade.co.in

Year of Publication 2023

International Standard Book Number-13: 978-81-19199-48-8

CONTENTS

Chapter 1. Basics of Computer Architecture 1

— Sunanda Das

Chapter 2. Comparing Intra-LDS and Intra-Permute Performance 10

— Chandramma R

Chapter 3. Reliability & Performance Trade-off Study of Heterogeneous Memories........ 20

— Vanitha K

Chapter 4. Implementation of Cross Counters........ 30

— Sonal Sharma

Chapter 5. Building Blocks of Contemporary Computers........ 41

— Jagdish Chandra Patni

Chapter 6. Networks of Computers 52

— Mahesh T R

Chapter 7. Implementing Server Architecture 63

— Gaurav Londhe

Chapter 8. Classification of Computer........ 73

— Ramesh S

Chapter 9. Von Neumann architecture........ 81

— Rajesh A

Chapter 10. The Amdahl Law........ 94

— Shashikala H.K

Chapter 11. Risks of Instruction-Level Parallelism 104

— C R Manjunath

Chapter 12. Dynamic RAM (DRAM) Technology 112

— Sowmya M S

Chapter 13. Electronic Memory........ 122

— Krishnan Batri

Chapter 14. Bus Development 133

— N Sengottaiyan

Chapter 15. Classification of Parallel Bus 141

— Merin Thomas

CHAPTER 1

BASICS OF COMPUTER ARCHITECTURE

Sunanda Das, Associate Professor,
Department of Computer Science and Engineering,
Jain (Deemed to be University) Bangalore, Karnataka, India
Email Id- sunanda.das@jainuniversity.ac.in

System dependability has grown to be a major worry as technology becomes smaller. This is a result of the growing diversity in circuit properties seen in nanometer-scaled microsystems. These variances in semiconductor production show up as a rise in soft mistakes, timing problems brought on by accelerated circuit ageing, and a rise in device failure rates. This dissertation looks for software-based methods to identify, fix, and avoid such mistakes in memory and compute-related components. Software-based methods for error detection and recovery have a significant performance cost to the system as a whole. In this thesis, performance overhead associated with software-based error prevention is reduced. This article specifically suggests the use of fingerprinting and cross-lane instruction as two strategies for efficient software-based error detection in compute units. Fingerprinting uses hashing to merge several error detection events into a single event, while cross-lane instructions use low-latency register-level communication to facilitate error checking. Moreover, this thesis investigates Application-Specific Approximate Recovery to lessen the performance overhead of software-based recovery (ASAR). In order to lessen the performance cost of software-based recovery, ASAR compromises on output quality.

Variance impacts both memory and compute components. For memory units, we concentrate on strategies for proactive error avoidance. We concentrate on developing heterogeneous memory architectures because of the variety of memory components in use (HMAs). Several memory modules with various performance and reliability characteristics make up an HMA. Different error-correcting codes, various memory module types, and the effects of age may all contribute to these discrepancies. In order to lower the possibility of running into an error, this thesis focuses on techniques for placing and moving data objects throughout memory modules in an HMA system. Age-aware and vulnerability-aware data placement are two unique data placement approaches that are specifically covered in this paper. Whereas the vulnerability-aware approach determines how susceptible data in memory is to soft errors, the age-aware technique tracks the buildup of faults in various memory modules as they become older. With the help of this thesis, software-based error mitigation techniques may be effectively applied to both existing and future hardware.

Today's microelectronic devices are constructed employing process technology, which yields microscopic characteristics. Some of these characteristics, like the gate oxide thickness, are just a few atoms wide. Geometries and dimensions are becoming harder to regulate at this level. Circuit behaviours may vary noticeably due to even the smallest manufacturing variations, which are often seen in gate delays and power consumption. However, some of these variations are linked to the "age" of circuits, or the way that well-characterized material changes in semiconductor devices cause circuit delays to fluctuate with time. These variances ultimately lead to increased ageing (wear out) and deviations from the intended behaviour (faults) in microelectronic devices. The increase in circuit complexity and integration scale comes at the expense of the compute and memory units' increasing brittleness and error-proneness. In order to identify, correct, and avoid mistakes in compute and memory

components, this thesis investigates and presents some useful, affordable software-based solutions. We specifically go through methods for lowering error detection costs for computing units. We also provide computing units an application-specific fault recovery system. Lastly, we suggest an age- and vulnerability-aware data placement to guard against deterioration and flaws, respectively, in memory units.

Scaling of Technology

Higher degrees of integration and the associated benefits of Moore's Law of scaling are made feasible by technology scaling, or the movement of designs to upgraded processing nodes. The hardware elements, such as transistor devices and the circuit components created from them, however, no longer perform as exactly as anticipated by circuit modelling tools with strict tolerances on their timing or power consumption as a result of rising manufacturing variance. In fact, performance and power uncertainty are already present in new devices. Variability is the term used to describe these effects collectively and generally. Variability appears as faults in memory and computation. The use of conservative voltage and frequency guard bands by device and system designers masked the impacts of variability-induced uncertainties and ensured error-free operation. Such guard bands really take up over 40% of the cycle time and use a lot of power when sleeping, thus decreasing the advantages of modern process technology nodes. Other methods of reducing the impacts of variability must be developed since extensive guard banding against those effects would have an unaffordably high performance cost.

Despite the fact that hardware-based error mitigation approaches find and fix faults with a worst-case cost of one cycle, they are rigid, expensive in terms of how they affect circuit size, and they prolong the time to market. The flexibility and speed with which software-based approaches may be implemented comes at a cost, however, in terms of performance and energy efficiency. So, in order to make software-based approaches practicable, we need to find ways to lessen the overheads they provide. As new computing and memory architectures lack robust hardware RAS (Reliability, Availability, and Serviceability) capabilities, low-cost software-based approaches are especially crucial. In this thesis, we investigate affordable software-based RAS methods.

New Developments in Memory and Computing

The semiconductor industry has profited over the last 30 years from two significant phenomena: Moore's law and Denard scaling. Every 1.5–2 years, the former makes it possible to integrate a greater number of transistors (usually 2X) onto a constant die size, while the latter makes it possible to employ more demanding operating conditions than the preceding generation. Faster execution is possible at the same power level or less thanks to aggressive operating point. The availability of more transistors has allowed designers to include more complex microarchitectural features, such as deeper instruction pipelining, branch prediction, prefetching, and out-of-order execution, that further exploit instruction level parallelism (ILP) and boost speed. These developments have allowed the semiconductor industry to increase single thread performance during the last three decades. Nevertheless, the physical restrictions of thermal management, growing guardbands, and increased leakage current all act as barriers to future scaling, thus stopping the effort to improve operating conditions.

Instead, the focus of the community for computer design has switched to creating parallel computation architectures (multicores and Graphics Processing Units). These computers improve performance by operating many threads concurrently as opposed to one thread more quickly. Peak memory bandwidth is what restricts the performance of parallel computers

when numerous threads are accessing memory. Moreover, sharing cache space across concurrent threads lowers the effective cache size per thread. High memory traffic is the result of running concurrent threads and having less effective cache capacity. Parallel architectures are progressively using high bandwidth 3D die-stacked memory units to accommodate growing memory traffic.

The newly developed 3D die-stacked DRAM technology has shown to reduce the difficulty with memory bandwidth. In particular, 3D die-stacking technology permits the use of through-silicon via (TSV) to stack one or more memory dies on top of one another. Internal datapaths are shortened as a consequence of TSVs, which lowers capacitance and active power. Moreover, 3D die-stacked memory requires a bigger bus width, increasing bandwidth. Comparing 3D die-stacked memory to traditional DDR memory, the bandwidth increase is 2X to 8X. The market now offers 3D die-stacked memory from the majority of the main semiconductor manufacturers, such as High Bandwidth Memory (HBM) from AMD, Wide-IO2 (WIO2) from Samsung, and Hybrid Memory Cube (HMC) from Intel.

Reliability Emerging-Architecture Issues: Transistors and circuits become more prone to failures as technology advances, and as was previously said, hardware-based fault mitigation techniques add complexity. In order to identify, recover from, and avoid hardware errors in developing architectures, we suggest and investigate software-based approaches. We focus on new parallel computing and memory architectures, including Graphic Processor Units (GPUs) and 3D die-stacked memory.

Architecture of the Computer's Reliability: Applications for speeding up video and image processing have grown to be quite popular with graphics processing units (GPUs) (graphics workloads). As graphics workloads are inherently error-tolerant, rigorous reliability requirements are not what drive the GPU design roadmap. Yet, studies have demonstrated that GPU parallelism may be leveraged to speed up a range of high-performance and safety-critical computing tasks. More applications, especially those that need absolute precision, such as avionics and driverless cars, are now possible with GPUs because to technology scalability and increasing form factors. We suggest using redundant threads to discover faults since parallel threads use the majority of the computer chip space. We investigate Redundant Multithreading (RMT) specifically on a GPU. A redundant copy is executed as part of the RMT error detection process, and the results are compared. When there is a mismatch, an error is signalled, and potential error recovery strategies may be used.

Similar to other software-based error detection methods, GPU RMT has a significant performance penalty. By conducting tests on a real GPU, we demonstrate that the majority of the GPU RMT performance cost is from synchronizing original and redundant copy rather than redundant copy itself. In this research, we suggest the use of cross-lane instructions and fingerprinting to decrease the synchronisation cost of GPU RMT. Cross-lane instructions provide thread-level synchronisation via register-level communication, while fingerprinting hashes many synchronisation events into a single event. We put our methods into practise as a transformation step inside the compiler that can be used to activate or disable software-based error detection during compilation.

We also look at inexpensive software-based error recovery methods. Extending Redundant Multithreading to Triple Module Redundancy (TMR) is a simplistic approach to recovery implementation. As implied by the name, TMR employs majority voting to recover from errors and runs three copies of the same thread simultaneously. Nevertheless, there is a non-linear rise in performance overhead due to the cost of synchronising three copies. Although error detection must always be active, error recovery must be used when an error does occur. Hence, we research a checkpoint and recovery system that is gradual. This makes it possible

to separate the overhead for recovery from that for detection. Recovery is often achieved by re-execution. Nevertheless, recovery by re-execution is not required for many Recognition, Mining, and Synthesis (RMS) applications. We provide low-cost Application-Specific Approximate Recovery in this thesis (ASAR). Software Recovery Blocks (SRB), a well-known programming feature that allows a programmer to specify application-specific error recovery code, are used to build ASAR.

Architecture of Memory That Is Reliable: According to the 2009 ITRS roadmap, the chip real estate used by memory systems is expected to rise by 50x by 2024. As a result, memory problems are multiplying and becoming a major issue, particularly in large-scale systems. Large-scale systems nowadays use memory chips from a variety of standards and manufacturers, resulting in a heterogeneous memory architecture. Several memory modules are used by memory systems to meet their bandwidth, latency, and capacity needs. One particular HMA system makes advantage of developing 3D die-stacked memory for bandwidth and traditional DDRx for capacity.

The dependability issues with memory systems have been further worsened by HMAs employing 3D die-stacked memory. Die-stacked memories are eight times denser and more susceptible to recent failure modes, such flawed through-silicon vias (TSV). Due to these elements, 3D die-stacked memories are more prone to errors than traditional off-package DDRx memory. Due to cost and complexity restrictions, die-stacked memory also uses weaker error correcting codes. Modern research uses a variety of cutting-edge methods to find and fix mistakes in die-stacked memory. Unfortunately, the complexity, energy use, and performance of these methods are high. As a result, we suggest two proactive age- and vulnerability-aware data placement approaches for HMA systems.

Memory modules' heterogeneous manufacturing processes, suppliers, and error-correction capabilities are to blame for their varying ageing rates. We show that performance-focused data placement is harmful to overall system dependability when memory systems become older and develop defects. In order to maintain the consistent functioning of a long-running system, we advise using an age-aware data placement strategy that checks the condition of available memory modules and chooses a memory module.

Moreover, we classify the data of various workloads depending on their level of susceptibility and hotness and examine the memory footprint of such workloads. We use access (read/write) counts to calculate the hotness of the data. We use the well-known Architectural Vulnerability Factor (AVF) study to determine vulnerability. The likelihood of experiencing an observable program error ascribed to a particle hit is known as the AVF of a data item in memory. With the use of data vulnerability analysis based on AVF, we make the following contributions. We demonstrate that the availability of recent data does not always imply vulnerability. This attribute opens the door to creating data placement strategies that maintain an HMA system's performance and dependability. We suggest dynamic migration and profile-guided data placement procedures in an effort to lessen the susceptibility of the whole system to particle assaults.

Using the vocabulary below, we distinguish between faults, failures, and mistakes in this thesis. An inherent reason for a failure, such as a particle hit, a stuck-at bit, or wear-out, is referred to as a defect. Failure is a little departure from the expected behaviour. In instance, a failed bit cannot be successfully read from or written to. When reading from a failed bit, data other than the bit that was most recently written to it is returned. Error correction systems that are based on hardware or software may stop a failure from spreading to higher levels. A failure's symptoms include errors: Error-correcting systems in hardware or software may find and fix mistakes. As a result, mistakes may be further divided into silent data corruption,

detected uncorrectable errors (DUE), and recognized correctable errors (SDC). Silent data corruption might come from a mistake that error correcting algorithms are unable to detect.

Permanent vs. Temporary Faults: Transient and persistent faults are the two major categories for faults. Temporary errors cause a bit to momentarily deliver incorrect data up until the bit is replaced. Bits that have permanent errors often yield the wrong value. The frequency of these failures may be accelerated by a number of circumstances, including voltage dips, particle strikes, process, ageing, and temperature. In this dissertation, we present software-based methods for preventing both temporary and irreversible errors in new compute and memory architectures.

Dissertation Organization: For upcoming computer architectures, software strategies targeted at lowering the overhead of error detection. We talk about redundant multithreading (RMT) for Graphics Processor Units in particular (GPUs). In order to reduce the performance overhead of error detection, we first develop cutting-edge RMT algorithms on a GPU and divide it into two main categories: redundancy and synchronisation overhead. We demonstrate that the performance overhead of GPU RMT is dominated by synchronisation. To decrease the synchronisation cost of GPU RMT's fingerprinting and cross-lane operations software strategies to recover from hardware errors, we introduce two unique compiler modifications. We first provide a hardware design that communicates software layer error information. The trade-off between performance and output quality for different software-based error recovery strategies is then discussed. We specifically develop the software-based recovery strategies I ignore error, (ii) fully recover through rerun, and (iii) partly recover via approximation execution. We also utilise a hybrid recovery approach that combines approximation- and rerun-based recovery. Using the well-known programming feature known as Software Recovery Blocks (SRB), we allow the programmer to contribute Application-Specific Approximate Recovery (ASAR) code.

new architecture for heterogeneous 3D die-stacked memory (HMA). We concentrate on a particular example of an HMA system that combines both conventional DDR memory and 3D die-stacked memory to fulfil bandwidth and capacity requirements. On the HMA system, we undertake a reliability vs. performance trade-off analysis. The issue that an HMA system that simply optimises for performance may have degraded dependability over time owing to the buildup of permanent defects is shown in the chapter. Lastly, in order to guarantee the dependable functioning of old systems, we provide an age-aware access rate control technique.

utilising the well-known Architectural Vulnerability Factor (AVF) methodology to calculate data vulnerability for data in memory Data vulnerability assesses the possibility of running into a software problem that may be traced back to temporary issues with memory-resident data. The study of data vulnerabilities for a broad range of multicore workloads is covered in this chapter. Based on their susceptibility to momentary mistakes, memory data are divided into low- and high-risk categories in the study. In order to safeguard memory units from transient failures, we also provide dynamic migration methods and profile-guided static data placement approaches.

Synchronization GPU Reduction Multitasking with Redundancy (RMT)

By duplicating computation at the thread level, Redundant Multi-Threading (RMT) offers a potentially inexpensive way to raise the dependability of developing computing (GPUs). The substantial performance overhead of RMT is caused by both the execution of duplicate threads and the synchronisation costs incurred by the original and redundant threads. If global memory is used to accomplish the synchronization, the overhead of inter-thread

synchronisation may be very large. In order to minimise the synchronisation cost for RMT on GPUs, this chapter introduces cutting-edge compiler approaches that use fingerprinting and cross-lane operations. Cross-lane operations allow thread-level synchronisation via register-level communication, while fingerprinting combines many synchronisation events into one event by hashing. As compared to the most recent GPU RMT solutions on actual hardware, this chapter demonstrates that fingerprinting results in a 73.5% reduction in GPU RMT overhead while cross-lane operations cut the cost by 43%.

Multitasking with Redundancy (RMT)

Using redundant copies of a thread to compare results, redundant multithreading (RMT) is a transitory error detection approach. When there is a mismatch, an error is signalled, and potential error recovery strategies may be used. Both the redundant and the original threads must synchronise in order for the comparison to be possible. We concentrate on GPUs because they provide a great platform for increasing parallel processing and since high-performance embedded computing uses them often. More domains can now use GPUs, including those that need absolute precision, like avionics and driverless cars, thanks to technology scalability and increasing form factors. Particularly, we concentrate on GPU built within Accelerated Processor Unit (APU). On a single SoC, an APU combines a high-performance GPU with a low-power CPU. The form compact, power efficiency, and visual capabilities needed for portable embedded devices are all well-balanced with APUs.

Executing the original redundant thread pair either (a) in lockstep inside a single group (Intra-Group) or (b) separately in different groups (Inter-Group) are two ways to implement RMT on a GPU. The former has less performance overhead whereas the latter gives better mistake coverage. Intra-Group RMT uses low-latency local memory that is exclusively accessible to threads within a group to accomplish synchronisation. Using somewhat slow global memory that is accessible across the whole GPU, Inter-Group RMT provides synchronisation. These GPU RMT approaches share performance overheads with the majority of software-based error detection strategies, making them only practical for the most performance-tolerant safety-critical applications.

Costs associated with redundant multithreading

We break down the performance overhead of the most advanced RMT methods into redundant computation and synchronisation overheads, to better understand the performance bottlenecks. For our benchmark suite (described below), the overall performance overhead for Intra- and Inter-Group RMT are 65.4% and 1785.9%, above the baseline execution time, respectively. The complete kernel runtime of the original, unaltered programme constitutes the baseline execution time. The performance penalty is dominated by the synchronisation overhead component, which accounts for 84.1% of the overall overhead for Intra- and 1726% of the overall overhead for Inter-Group. Not just for Intra- and Inter-Group GPU RMT, but also for several CPU-based error detection techniques, synchronisation has been found to be a bottleneck. Soft mistakes may be detected using software-based redundancy approaches at the instruction, process, and thread levels. Error recovery overhead is independent of detection overhead; if recovery is carried out via incremental checkpoints, it is independent of both. Moreover, error recovery is only used when an error is identified, while error detection continues continually. We concentrate on lowering mistake detection overhead in this effort.

By using cutting-edge compiler methods, we propose and show a decrease in the synchronisation cost for RMT, extending its applicability to performance-critical applications. By executing apps on actual hardware, we assess our work. The contributions made by this chapter are as follows:

quantitative study to differentiate between redundant calculations and thread synchronisation in the performance cost of RMT. For Inter-Group RMT, we further divide the synchronisation cost into locking and communication. Our findings indicate a 73.6% decrease in transmission overhead and an 80.4% decrease in locking overhead for RMT.

Intra-Permute is a fresh Intra-Group RMT optimization. Intra-Permute makes advantage of newly developed cross-lane operations, which have been proved to be an effective technique to transfer data across threads executing in the same (lockstep) lane. We demonstrate that Intra-Permute lowers the overhead of Intra-Group RMT by 43%, from 65% to 37%.

Inter-Fingerprinting is a novel fingerprint-based Inter-Group RMT optimization (Inter-FP). Inter-FP utilises hashing to merge numerous synchronisation events into a single event. We put into practise a lightweight XOR hashing and contrast it with a reliable CRC32 hashing. Our decrease in Inter-Group overhead is 73.5%.

We provide our methods as a compiler transformation pass that may be used to activate or disable software-based error detection at the time of compilation. Our methods are applicable to a variety of GPU architectures that allow cross-lane operations.

Processor Unit for Graphics (GPU)

In this part, we first outline the GPU execution paradigm and then go into great depth into the most recent GPU RMT approaches.

Model for GPU execution

The background of our work on RMT compiler changes is AMD's GCN GPU architecture. It may be used with other architectures as well, however. In this part, we define several concepts unique to RMT and GPU terminology that is in line with the selected architecture.

The only kernel that can execute on the device is Work-item, which is a GPU thread.

A single SIMD unit executes 64 work-items collectively under the name Wavefront in lockstep.

Workgroup is a construct used at the programme level made composed of one or more wavefronts. Work-items in a workgroup may exchange information through local scratchpad memory.

The collection of hardware structures protected by computation replication is referred to as the "Sphere of Replication" (SoR).

The Compute Unit (CU), is the central component of the GCN design. Four 16-wide Single Instruction Multiple Data (SIMD) units make up each CU. A wavefront with 64 work-items may be executed in lockstep over 4 cycles by a SIMD unit. Each CU features a 64-kB register file with a low-latency scratch-pad memory and a 64-kB local data share (LDS) that can handle up to 256 64x32-bit vector register files (VRF). Four 3200-byte scalar register files and one scalar unit (SU) are also included in the CU (SRF).

C++AMP

A C++-based single source programming language created for heterogeneous computing is called C++AMP. Lambda functions are code areas that C++AMP encodes so they can be executed on accelerators, such as GPUs. These lambda functions are converted into accelerator calls by the compiler, which also generates the accelerator device's binary, inserts the required runtime calls, and starts the kernel. C++AMP programming for heterogeneous systems takes away the difficulties of device startup and runtime management.

Adaptive System Architecture

A set of requirements known as Heterogeneous System Architecture (HSA) enables the coexistence of CPU and GPU on the same chip. With unified virtual memory, HSA-enabled devices may share system memory, drastically lowering CPU and GPU communication latency. HSA-enabled devices are the primary target of HSA compilers and driver technologies like the Heterogeneous Compute Compiler (HCC). The HCC compiler is an integrated compiler that enables the compilation of C++AMP code from a single source into GPU kernel binaries and CPU host code. The HCC compiler produces kernel code using the HSA Intermediate Language using the High Level Compiler (HLC), which is the main GPU HSA compiler (HSAIL) The optimizer tool (opt) component houses RMT updates to the HLC compiler. The optimizer processes the GPU kernel code produced by the frontend clang++ compiler using the Low Level Virtual Machine (LLVM) intermediate representation (IR). To prevent our changes from being removed from the final binary during IR optimization, the RMT compiler transformations are performed after all IR optimizations. GPU kernel code that has undergone our RMT changes is output in HSAIL at the conclusion of the compiler process. Our RMT compiler changes may be ported to any HSA-enabled accelerator using the finished HSAIL GPU kernel code, regardless of the accelerator's architecture. For our RMT compiler transformation, the HCC compiler was chosen to address trends towards the programmability of heterogeneous systems.

Operations Across Lanes

Every HSA-enabled device may utilise our Intra-Permute compiler transformation since it will guarantee that the cross-lane operations will be executed (natively or via software emulation) in accordance with the HSA standards. Cross-lane operations may be natively executed on NVIDIA Kepler and AMD GCN3 GPUs using the shuffle and permute instructions, respectively. These instructions allow for effective work-item (WI) communication at the register level inside a wavefront. A work-item that is performing permute might specify the wavefront's source (pull semantics) or destination (push semantics) work-id item's a cross-lane instruction pull semantics scenario where all even WIs inside a wavefront are drawing data from the nearby odd WIs.

Existing GPU RMT is a reliability strategy that uses duplicated computation rather than hardware duplication to find hardware flaws. The Sphere of Replication is claimed to include hardware protected by RMT (SoR). The original and redundant work-items employ inputs that are reproduced when data is introduced into the SoR, such as from global memory. Up until the execution exits the SoR, the original and copied calculations proceed independently of one another. To facilitate output comparison, every state change outside the SoR need synchronisation. The original copy receives the outputs produced by the duplicated copy, and these are checked for errors. A mismatch indicates a detected mistake, while matching outputs show error-free operation. RMT is appealing for hardware components with high thread counts and built-in latency tolerance, including GPUs and the vector engine in APUs. The first RMT discrete GPU implementation was released. Work-items are duplicated by intra-group RMT inside a wavefront. In the same SIMD unit, the original and redundant work-items run in lockstep. The original work-item compares the communicated data with its local data to check for mistakes every time a store leaves SoR. The vector register file and SIMD units are inside the SoR, as shown by the dashed line the latest technology The "Intra-LDS" moniker refers to the intra-group RMT algorithm's use of local memory (LDS) as a communication channel.

The whole workgroup is replicated using inter-group RMT. The original and redundant work-items across several workgroups do not guarantee execution in lockstep. The solid line in

how Inter-SoR is a superset of Intra-SoR and offers more error coverage than Intra-Group RMT. On top of Intra-SoR, Inter-SoR adds instruction fetch, decode, scheduler, scalar unit and register files, and local memory. With LDS, work-items from several workgroups cannot synchronise. Every time execution exits Inter-SoR, Inter-Group RMT executes the synchronisation procedure using a global memory store, or buffer. This paper uses the term "Inter" to refer to the cutting-edge Inter-Group RMT.

Limits of Current GPU RMT

Intra-LDS RMT first requires extra LDS memory for LDSBuff. As LDS memory is limited, it could become a bottleneck for applications like tiling matrix multiple, which utilise it to optimise performance. Second, two special instructions, memfence(acquire) and memfence, are used to enclose the data transmission from WIB to WIA utilising LDSBuf (release). Correct memory ordering and visibility are specified via the memfence instruction. As long as memory is constant for the work-item, memory instructions inside a work-item may be rearranged. Moreover, a value could exist in the L1 cache before being committed to the LDSBuf. Memfence() instruction should be used, but, if a memory region has to remain constant across several work-items inside a workgroup. Reordering is prohibited by memfence's semantics, which also guarantees proper memory visibility for various work-items inside acquire-release borders. the error check and commit the protected store to memory are only carried out by WIA. Divergence should be avoided since it may lead to a decrease in GPU performance. Before to synchronisation, Inter RMT needs slower global memory. Second, a leading and trailing work-item is created since the original and redundant work-items were not executed in lockstep. Additional locking overhead is required during leading and trailing work item synchronisation in order to guarantee proper memory ordering on the shared global memory. Two main parts make up the synchronisation overhead: locking (L) and communication (C). The locking component implementation calls for an extra global memory allocation to hold the lock state for each work-item pair (lock buffer is not shown in the flowchart). To determine if it may write the data (value and address for Inter) to GMBuf, the acquire lock block in the leading work-item reads the lock value from the lock buffer. Similar to this, before loading the data (value & address) and performing the error check, the wait lock in the trailing work-item checks the lock value to verify if the data was written by the leading work-item. Implementing locking results in even more global memory traffic, which raises synchronisation cost.

GPU RMT Hashing with Intra-Permutation. For the getHash function, XOR and CRC32 are implemented. XOR offers a simple method for capturing certain single bit flips per hash, but not all of them. With XOR, the standard Hamming distance is 2, which may detect single bit mistakes. Two single bit faults in the same bit location in two hashed data items are unlikely to be hidden, nevertheless. We also use computationally costly CRC32-based hashing to account for multi-bit mistakes. In order to detect all single bit mistakes among five hashes of 32-bit length, we employ the IEEE 802.3 CRC32 polynomial function, which has a Hamming Distance of six for 268-bit data length. CRC32 has a very low chance of colliding, i.e. As the redundant work-item duplicates the hashing calculations, any hashing errors are likewise protected to the same extent as the rest of the kernel code. The terms FPNXOR and FPNCRC32, where FPN stands for fingerprinting with threshold N, will be used to designate to the XOR and CRC32 Inter-FP versions.

CHAPTER 2

COMPARING INTRA-LDS AND INTRA-PERMUTE PERFORMANCE

Chandramma R, Assistant Professor,
Department of Computer Science and Engineering,
Jain (Deemed to be University) Bangalore, Karnataka, India
Email Id- chandramma.cse@gmail.com

Intra-LDS and Intra-Permute RMT variant performance cost was normalised to the kernel's initial runtime. For every benchmark other than snap-c, Intra-Permute outperforms Intra-LDS. The amount of dynamic local (DLS) and global memory stores (DGS) present in a benchmark determines the possibility of lowering the communication cost using Intra-Permute — see Table 2.1.

The benchmarks with less dynamic storage, SM, SPMV, XSB, and mAR, don't benefit much from Intra-Permute. Yet, by lowering the communication cost associated with each dynamic store, MM, FFT, and CoMD with additional dynamic stores demonstrate a significant decrease in delay with Intra-Permute. In comparison to Intra-LDS, snap-c performs worse with Intra-Permute.

Intra-Permute modifies the instruction mix to include more VALU instructions while increasing register pressure. If the original kernel has mostly VALU instructions, then this might lead to less parallelism and a slowdown. Intra-LDS and Intra-Permute often take 1.65 and 1.37 times as long as the baseline, respectively. The average RMT performance overhead is decreased by 43% (43% for Intra-Permute) from 65% to 37%.

Comparing inter and inter-FP performance

Each benchmark's first bar depicts the slowing of Inter RMT. For instance, whereas FFT displays a 126x slowdown of the baseline duration, SPMV displays a 2x slowdown. The two main conclusions to be drawn from the Inter performance results are that a) the slowdown for Inter-Group RMT is 23 times worse than Intra-LDS RMT due to the use of global memory as a communication medium and the explicit locking mechanism to coordinate synchronisation between work-items, and b) the performance overhead varies significantly between different benchmarks. Show the slowing for the FP2-8XOR and FP2-8CRC32 RMT versions for each benchmark.

According to the results of the Inter-FP performance analysis, a) performance enhancements by raising the fingerprint- ing threshold (FPN) depend on the quantity of dynamic global memory stores (DGS) present in a benchmark (see Table 2.1), and b) the computationally demanding FPNCRC32 only exhibits a slight slowdown in comparison to FPNXOR. Unless unless stated expressly, the discussion that follows compares the performance of Inter-FP with Inter. Inter-FP enhances efficiency by minimising synchronisation events (Figure 2.1).

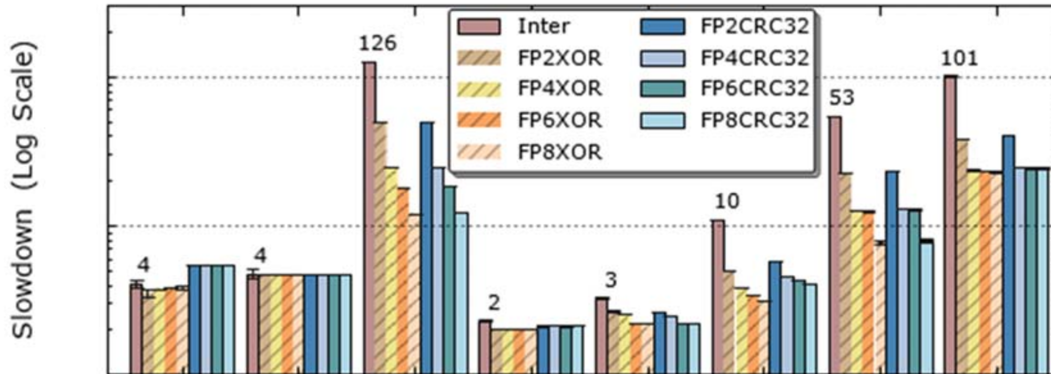

Figure 2.1: Illustrates the marginal increase in slowdown relative to FPNXOR.

Inter-FP is unable to decrease the number of synchronisation events for the benchmarks MM, SM, and SPMV using a single DGS. Even yet, Inter-FP may still save bandwidth by sending only one hash rather than the value and address that are sent with Inter. Due to the decreased communication bandwidth brought on by hashing, MM (only FPNXOR), SM, and SPMV exhibit improved performance. Benchmarks with a single DGS demonstrate that raising the fingerprinting threshold has no performance advantages (FPN). The performance of the benchmarks FFT, XSB, CoMD, MAR, and SnPC with multiple DGS does, however, improve with the addition of FPN. When 2, 4, 6, and 8 global memory stores are bypassed before synchronisation using FPNXOR, FFT with 16 DGS exhibits 60.7%, 80.7%, 85.8%, and 90.5% decrease in slowness. Similar to FFT, XSB and CoMD with Inter-FP versions exhibit comparable behaviour. mAR displays the decrease in slowness with FP2 and FP4 Inter-FP RMT. From FP4 to FP6, there is, however, no slowing decrease. This is due to the fact that there are eight DGS per work-item, which implies that there are two synchronisation events for each kernel call in FP4 and FP6. When comparing Hashing XOR to CRC32, the average performance overhead decrease with FPNCRC32 and FPNXOR is 73.5% and 74.25%, respectively. the FPN- CRC32 RMT variations' performance overhead adjusted to the FPNXOR RMT kernel's runtime. All of the benchmarks that are compute-bound are on the left, and all of the benchmarks that are memory-bound are on the right. We see that benchmarks that are compute-bound have a greater overhead for CRC32 calculations than benchmarks that are memory-bound. The cost of CRC32 for benchmarks that use memory is masked by memory latency. The CRC32 calculation overhead between MM and SM, which are at the opposite extremes of the compute- and memory-bound spectrum, differs significantly. In order to lower the overall memory traffic, MM leverages LDS memory in its improved version of the matrix multiply algorithm. The performance of the already compute-bound MM kernel is deteriorated by the extra CRC32 operations. Comparing MM with FPNCRC32 to FPNXOR RMT version, there is a 39.3% performance reduction. Comparing SM to FPNXOR, there is just a 0.2% performance decrease. Although CRC32 calculations generally have a 10.4% cost, memory-bound benchmarks (mAR, SPMV, XSB, and SM) only have a 1.8% overhead).

Breakdown of inter-FP performance: the associated hashing costs for Inter, FP2CRC32, and FP8CRC32. Inter RMT has a hashing overhead of 0. With FP2CRC32 and FP8CRC32, the hashing overhead is 21% more than the baseline. This is because FP versions do the same amount of reactive hashing. Inter, FP2CRC32, and FP8CRC32 each have locking overheads over the baseline run of 1001.2%, 439.4%, and 195.0%, respectively, as shown by the stacked bar. Comparing FP2CRC32 and FP8CRC32 to Inter RMT, the locking overhead is reduced by 56.1% and 80.4%, respectively. The overhead of redundant calculation for the

Inter, FP2CRC32, and FP8CRC32 RMT versions is shown in the second stacked bar from the bottom. For all three versions, redundant calculation cost is around 60% more than the baseline run.

Use of Energy (EU)

The GPU's energy use is measured in Energy Usage (EU). The region underneath the GPU power consumption over time curve is referred to as EU. Using CodeXL, we periodically sample power use at 50 ms intervals to estimate EU. The first two bars in the average EU for our benchmark suite with Intra-LDS and Intra-Permute RMT increases by 1.30x and 1.25x of the baseline EU, respectively. Compared to Intra-LDS, Intra-Permute resulted in a 16% decrease in energy overhead. The baseline average power is increased by 1.17x and 1.18x, respectively, for intra-LDS and intra-permute RMT. Compared to Intra-LDS, Intra-Permute has an average power that is 5% higher. Faster intra-thread communication made possible by intra-permute allows for more vector ALU instructions to be processed per clock and a marginal increase in average power. Thus, the shorter execution time is the main factor in the lower energy consumption. The basic EU served as the standard for the EU for the Inter and Inter-FP versions. FPNXOR and FPNCRC32 result in energy overhead reduction of 64.4% and 62.2% compared to Inter, resulting in an average EU for our benchmark suite of Inter, FPNXOR, and FPNCRC32 of 10x, 4.2x, and 4.4x of the baseline, respectively. The average power with Inter, FPNXOR, and FPNCRC32 is, respectively, 2.0 times, 1.76 times, and 1.83 times that of the baseline power. Contrary to their relative performance overhead, CRC32's EU overhead is consistently greater than XOR's across the majority of benchmarks. The memory delay may mask the performance expense of CRC32's additional calculations, but average power and EU still reflect this burden.

Software Recovery Blocks (SRB) are a programming concept that allow programmers to create application-specific error recovery code. ASAR stands for Application-specific Approximate Recovery. Rerunning the calculation or rejecting the incorrect one are the two recovery options. The user has two extreme operating points on the performance-quality trade-off curve: rerunning incurs a performance expense, and rejecting incorrect calculations could lead to decreased output quality. This paper suggests approximation recovery, which is especially helpful for applications using approximate computation, in order to take advantage of intermediate performance-quality trade-off points. The paper provides an SRB enhancement dubbed Application- Specific Approximate Recovery for such applications, which naturally tolerate failures (ASAR). In comparison to rerunning, ASAR speeds up six approximation computing applications by 3.8% to 29.9%. The chapter also suggests a hybrid recovery technique that enables a user to specify the output quality they prefer and to take advantage of the performance-quality trade-off curve at a finer granularity. Using a combination of ASAR and rerun-based recovery, hybrid recovery shows a 1.5%–11.6% speedup over rerun alone while retaining user-specified output quality.

Restoration Mechanisms

Both hardware- and software-level recovery techniques are used. Redundancy, circuit-level strategies, and non-trivial design-time approaches, which attempt to lessen architectural vulnerability factors, are only a few examples of hard- ware recovery solutions. Repetitive code execution, checkpointing & re-execution, and compiler-driven vulnerability reduction are a few of the well-liked software strategies. All of these methods have limitations; for example, software-only systems suffer from over 2-fold speed and memory cost due to the duplication of all computations required for error detection. The goal of hardware-only solutions, on the other hand, is to hide every problem and provide software the appearance of error-free operation at a significant overhead cost.

Tight Separation between HW and SW

The majority of hardware and software techniques maintain a distinct separation between the two, masking any hardware defects in the programme. Particularly for applications involving approximation computation, this rigid separation is costly and unneeded. The semiconductor business is greatly influenced by approximate computing workloads, such as search, multimedia, finance, and big data. Conceptually speaking, these programmes contain a vector of elastic outputs, thus even if execution is imperfect, the programme may still give usable output. Users may choose from a wide range of operating points on the performance-output trade-off curve provided by approximation computing applications by loosening the division between hardware and software.

Relax separation of HW and SW

Our strategy for dealing with mistakes brought on by variability is to include such processing into the current software handling methods. Hence, similar to exception handling, we suggest exposing hardware fault information to the programme. For instance, modern hardware may gracefully terminate programme by exposing a divide-by-zero or memory-access violation. Instead, we look for solutions to fix faults brought on by unpredictability to guarantee that the system keeps working. In order to handle variability-induced hardware faults, we expand and assess the usage of software recovery blocks (SRB). Hardware may be operating in an "unsafe" regime for the code areas contained by SRB as a result of insufficient guard bands, such as a lower voltage and/or higher frequency. The programme is informed of any ensuing calculation failures as part of the SRB semantics. In the event of a mistake, the runtime has three options: a) restart the code to guarantee 100% correctness, b) approximate the calculation to guarantee partial recovery, or c) discard sub-computations to entirely disregard the error. The application developer may choose one of the aforementioned recovery strategies (a, b, or c) or a hybrid recovery, which combines two recovery approaches, depending on the acceptability of the user-provided output and algorithmic limitations.

Applications that use approximation computation may thus be divided between code segments that need error-free execution and segments that can accept variable degrees of error. The former is what we refer to as crucial, whereas the latter is non-critical. To allow for unsafe mode of operation, the non-critical code section is encased within the SRB. Dual-voltage operation, adaptive DVFS and execution migration across multi-cores may all be used to implement the dangerous modes of operation. With this method, we may operate at lower power or speed execution by taking advantage of the dangerous zones. For instance, we might switch between safe and dangerous cores or modify DVFS settings at the SRB borders. We provide three contributions in this chapter:

We suggest extending Software Recovery Blocks (SRB) to Application- Specific Approximate Recovery (ASAR), which is especially well suited for programming in a language with exception handling capability. As an extension of the standard Try-Catch mechanism, which is a high-level programming construct, ASAR may identify hardware failures and provide approximate software recovery options.

We show that for six approximation computing applications, ASAR improves performance by 3.8% to 29.9% compared to rerun and output quality from 5.4 to 84.3 percentage points compared to discard. As a result, ASAR offers a user a midway point on the performance-quality trade-off curve.

We provide a hybrid recovery method that combines ASAR with rerun-based recovery and enables the user to choose an output quality criterion. We demonstrate that compared to a rerun with output quality better than the user-specified threshold, hybrid recovery speeds up

operations by an average of 1.5% to 11.6%. Application programmers may investigate the performance-quality trade-off curve at a more granular level thanks to hybrid recovery.

Software Recovery Blocks (SRB) Software recovery blocks (SRB) are a well-known programming paradigm in real-time embedded systems that allow application programmers to react to software errors. Programming that is fault-tolerant and exception handling are supported by traditional SRB implementation. This method of programming guarantees recovery from potential design flaws in software components. Software acceptance tests are used to find errors, and the programme uses the principal module to check for acceptability. The execution moves to the backup module if the main module fails the acceptance test. Segmentation and divide-by-zero errors may be gracefully handled by SRB in conjunction with hardware support. For instance, when a segmentation failure occurs, software tries to access a memory segment that is beyond of its acceptable range. Software thus lives smoothly while hardware sets up a trap. Software-induced errors, like segmentation faults, are ones that the hardware recognizes and helps the programme fix. Without the necessary support, a system may encounter a system crash that necessitates a reboot, data loss, and silent data corruptions.

We suggest using the Try-Catch method to expand the SRB technique for a system that exposes both hardware and software failures, much to the relax-recover mechanism developed. Timing errors may be corrected using application-specific knowledge, hardware error information, and hardware error information. Try blocks, which are part of the proposed Try-Catch system, run the main module in unsafe mode more quickly and with a larger chance of running into hardware faults. If a problem arises while the Try block is being executed, hardware causes an interrupt. Using the recovery module, which is integrated into the Catch block, the programme makes an effort to fix the problem. To guarantee error-free operation, the Catch block executes under secure operating settings. Rerun and discard are two software error recovery solutions that Kruijf et al. compare and contrast.

Restart Mechanism We execute the Try block code within the Catch block as part of the rerun or re-execution process until the hardware fault goes away. The multiple issue instruction replay is comparable to this software compensation mechanism. Rerun guarantees flawless output quality (QoS). Rerun-based recovery, however, incurs an execution time overhead. The overhead for WordCount (WC), K-Means (KM), and A2Time (A2T) with a single repeat. The overall execution time is shown on the y-axis and is normalised to the duration of error-free execution on the x-axis. The x-axis displays the rate of Try block failure. With a worst-case overhead for A2T of 60% and an average overhead of 27.7%, execution time overhead rises as failure rate does as well.

Throwaway Mechanism Sub-computations by an erroneous Try block are discarded in the discard process. The number of dropped sub-computations is encoded into the Catch block, which may be utilised to modify the outcome. The output quality (QoS) significantly degrades despite the discard mechanism's lack of recovery costs. Our preliminary analysis shows that the discard method yields the highest performance at the expense of perhaps unacceptable QoS deterioration. Even a 1% mistake rate causes a 15% QoS loss for A2T. We see a minimum QoS decline of 44% and a maximum of 27.33%. Try-Catch may be implemented directly using the restart and discard techniques. Yet, these processes function at the performance-quality trade-off curve's two extremes. For applications requiring approximation computation, our modification to the software recovery block, which is covered in the next section, offers a software programmable method to take advantage of intermediate performance-quality trade-off points.

Estimated Recovery via Application

Using approximation recovery, we expand the SRB technique to investigate intermediate performance-quality trade-off points. The application's algorithm determines whether to employ a certain approximation recovery strategy, such as sampling, interpolation, or reuse. In order to give an alternative for approximate recovery, we suggest and assess Application-Specific Approximate Recovery (ASAR). For approximation computing applications that can function successfully by sacrificing output quality for performance, ASAR offers a technique that sits in the middle of the two extreme recovery choices, repeat and discard.

A critical component and a non-critical section make up the programme execution in approximation computing. The setup code, configurations, system calls, and I/O activities make up the majority of the crucial component. The important code parts are not ideal candidates for software-based recovery because they cannot tolerate faults. So, in order to assure error-free operation, crucial code parts must be performed in safe mode. The side-effect-free sub-computations (idempotent areas), which mostly consist of hot code regions, or naturally error-tolerant floating point operations, should make up the non-critical code sections. Various compiler-based automated strategies may be applied to assure idempotent property for non-critical code portions. While using the rerun and discard recovery strategies, the impotence of non-critical sections is equally crucial. The Catch block uses rerun (paying the maximum recovery cost) or discard to recover in the event of an error (suffering maximum QoS degradation). As comparison to the rerun and discard techniques, an approximate software implementation may minimise both the recovery cost and QoS deterioration since non-critical code parts do not need to be exact.

Code that is critical vs. non-critical Only if applications spend a significant percentage of their time in non-critical regions are performance and energy advantages from execution under dangerous operating circumstances for non-critical code sections achievable. The dynamic ratio of critical to non-critical code, normalised to the overall execution time for eight applications. WordCount (WC), K-Means (KM), SOR, MonteCarlo (MC), Histogram (Histo), and PageRank (PR) are six of the eight programmes that spend 90% of their time in non-critical code regions. For WordCount, K-Means, and Histogram, comparable ratios of crucial vs. non-critical code parts. FIR and A2Time (A2T) respectively spend 75% and 30% of their time in non-critical regions. Before each call to a non-critical code segment, A2T does substantial setup calculations, and FIR comprises instructions that organise coefficients. Eight programmes, on average, spend 86% of their entire execution time in non-critical areas, according to our analysis. So, for certain applications, speeding non-critical areas through hazardous modes may yield in performance and energy advantages.

An error could occur when the non-critical code section is running in a risky environment. Rerun-based recovery, which incurs a significant recovery overhead, reruns the non-critical code within the Catch block under safe operating circumstances in order to recover from the mistake. We provide application-specific approximation recovery to shorten the time it takes for software to recover (ASAR). To develop ASAR for all eight applications utilised in this research, we employ sampling (samp), interpolation (inter), and reuse. The core ideas discussed in this work are not limited to the aforementioned approximate recovery methods; an application developer is free to use any other approximate recovery method of their choosing. Try-Catch blocks in our ASAR implementation may be seen as process nodes from the perspective of a common data-flow programming paradigm. Input tokens are used by Try-Catch, and output tokens are generated. Try and Catch are known as stateless data processing nodes because they do not store any data while the operation is in progress. The estimated recovery overhead normalised to rerun-based recovery of non-critical code section.

The non-critical code area is roughly implemented as lower the bar quicker.

Sampling by ASAR (ASARs). sampling in ASAR. The process node foo, which requires four input tokens, is executed by the Try block. Two input tokens are used in the example as the sample for the Catch block. Sampling is a frequently used approximation method for databases and multimedia.

We use the sampling-based approximation method for the WC and MC applications. Using approximation through ASARs, the first two bars in a 50% decrease in the execution time of non-critical code sections. As a result, when an error occurs, the Catch block employing ASARs recovers a failed Try block in half the time as compared to ASAR Interpolate (ASARi). Attempt blocks convert an input token stream (i-1, I and i+1) into an output token stream (O[i-1], O[i], and O[i+1]). If the Try "O[i]" block for a process node fails, the execution shifts to the Catch block. The Catch block utilises a user-defined interpolate function (inter) that activates the next process node by using the previous output token, O[i-1]. Try using O[i+1] to utilise the subsequent output token. In order to estimate O[i], the simplest interpolate function averages O[i-1] and O[i+1]. DSP algorithms that work with continuous time-domain signals are ideally suited to interpolation-based approximation. For DSP accelerators, Whatmough et al. provide hardware-based interpolation utilising pipeline lookahead. To speed up software-based recovery for A2T and FIR, two DSP applications utilised in this work, we construct and test ASARi. According to the second and third bars, utilising ASARi for A2T and FIR correspondingly, the non-critical code section's execution time was reduced by 40% and 83%, respectively.

Reusing ASAR (ASARr). Every time a Try block successfully processes input I an element of the reuse buffer is created for the next process node, and the outcome of the most recent iteration is saved there (RB[i]). The appropriate value from the reuse buffer is used to assign the output token O[i] in the event of an error. Applications with a high degree of input locality, such as multimedia benchmarks and iterative benchmarks, may benefit from reuse-based recovery. As indicated by the final four bars, we construct reuse-based approximation recovery for four applications, including two iterative (KM and PR) applications and two applications with strong input locality (SOR and Histo). K-Means is a clustering algorithm in which an input point in space, i is passed through the Try block, and the output, O[i], is a label for one of the available clusters. The K-Means method reassigns each point to the nearest cluster throughout each iteration. The Catch block has the ability to reassign the point from the previous iteration to the cluster in the event of an error. Reuse is implemented by PageRank similarly to K-Means. The reuse buffer stores the output token of the closest input for SOR and Histogram. The execution time of non-critical code sections using ASARr is reduced by 82% and 26% according to K-Means and SOR, respectively. When using ASARr, the execution time reduction for Histogram and PageRank is negligible or negative. The original implementation of the non-critical code region for PageRank and the histogram uses the same amount of instructions as reading from the reuse buffer in terms of execution time. Therefore, replacing rerun, which executes non-critical code in the Catch block again, with approximate recovery won't shorten the time it takes for Histogram and PageRank to recover. Throughout the remainder of the chapter we will explore and assess the first six applications ignoring Histogram and PageRank.

Hybrid Recovery (ASAR+Rerun)

ASAR delivers decreased recovery time compared to rerun and greater output quality relative to discard. Using ASAR to recover failed Try block provides user with an alternative option in between two extremes: rerun (worst performance, best quality) and discard (best performance, worst quality) (best performance, worst quality). ASAR results in faster

recovery times relative to rerun and improved output quality relative to discard. However, it only provides one more operating point on the performance-quality trade off curve. Additionally, using only approximate recovery may also result in an unacceptable output quality. Hence, we propose a hybrid recovery mechanism which uses both rerun and approximate recovery via ASAR. The ratio in which ASAR and rerun are triggered is called approximation ratio and is selected using quality of service model.

Quality of Service Model (QoSmod)

The QoS requirements are defined based on the quality of output or timing deadlines. To meet the QoS requirements, a model derives rules for selecting between rerun and ASAR block. In other words, the QoS model assists runtime determine approximation ratio in order to meet the desired QoS requirement. The following subsections, we describe the details of the QoS model generation and utilisation

QoS Model Generation

The upper dashed block in encloses the Qos model generation process. We generate QoSmod by executing an application for a wide range of Try block failure rates (f I and approximation ratio (r j) (r j). The failure rate f I represents percentage of Try blocks which fail due to unsafe operation and approximation ratio r j determines how many of failed Try blocks recover approximately via ASAR vs. rerun. We run experiments for each pair of (f I r j) on training inputs. For each experiment, the output is compared with the golden output. The golden output is the output of error-free execution (f I = 0). The final output of the QoS model generator is a discretized table QoSmod with QoS values for each combination of f I and r j. We use a coarser granularity Try block failure rate and approximation ratio to reduce the profiling time one-time QoSmod generation.

QoS Model Utilization

The runtime takes as input the generated QoSmod and user specified QoS threshold (QoSthd), as shown by the lower dashed block. The runtime failure rate can be estimated using standard hardware monitor detectors and/or hardware error models. For the estimated Try block failure rate and user specified QoS threshold we select an approximation ratio to ensure observed QoS greater than QoS threshold (QoSthd). Our results confirm that QoS model ensures QoSobs is always greater than QoSthd for test inputs (different from training inputs) and unseen finer-granularity failure rates. Thus, using QoSmod and hybrid recovery enables a user to specify an acceptable output quality threshold and explore various points on performance-quality trade-off curve.

Application-Specific Approximate Recovery (ASAR)

The performance and output quality for six applications using ASAR. The x-axis represents the rate of Try block failure, the left y-axis shows the execution time, normalised to the runtime of error-free execution, and the right y-axis shows the output quality using the QoS metric. The lines on each subgraph show the normalized execution time measurements. Approximate recovery via ASAR performs better than rerun-based recovery for all six applications. The average reduction in recovery overhead with ASAR relative to rerun is highest for Word Count, K-Means, MonteCarlo, and FIR. These applications spend substantial amount of their time in non- critical code region so the impact of ASAR is more pronounced. A2Time and SOR show only 3.8% and 7.5% reduction in recovery time relative to rerun. The gap between ASAR and rerun performance widens with the increase in the rate of Try block failure because at higher Try block failure rate faster approximation is employed more frequently. The reduction in performance overhead ranges from 3.8% to 29.9%. For all six applications the QoS loss increases monotonically with the increase in Try block failure

rate except for applications which are inherently random. MonteCarlo, an application that employs a randomised algorithm, computes the value of π by randomly choosing points in a two dimensional plane. The inherent random nature of the application attributes to the non-monotonicity in the QoS loss. We observe a maximum QoS loss of 38.8%. The average and maximum QoS loss and performance improvement relative to the rerun method for each application using ASAR are also listed QoSmod for seven coarser-granularity Try block failure rates f I ∈ [1%, 5%, 10%, 15%, 20%, 25%, 30%]. This range of failure rate is representative of variability-induced hard- ware failures (Table 2.1).

Table 2.1. Average and maximum QoS loss and performance improvement for ASAR and Hybrid recovery.

Benchmark	ASAR				Hybrid (ASAR + Rerun)			
	QoS Loss (%) Avg	Max	Performance Improvement (%) Avg	Max	QoS Loss (%) Avg	Max	Performance Improvement (%) Avg	Max
WordCount (WC)	19.6	38.8	24.8	53.6	12.2	21.3	11.6	21.7
MonteCarlo (MC)	15.7	31.1	13.7	26.5	5.7	12.4	6.3	15.5
SOR	19.8	22.7	7.5	14.4	18.8	20.1	3.9	4.1
K-means (KM)	12.2	21.3	29.9	69.5	6.2	13.0	8.1	16.9
A2time (A2T)	5.9	9.8	3.8	6.2	5.9	8.6	1.5	2.2
FIR	3.0	6.1	5.7	18.8	2.9	4.7	5.8	9.0

Qo Smod Hybrid Recovery (ASAR + Rerun)

Compared to rerun-based error recovery, ASAR dramatically lowers error recovery overhead (6.2%-69.5% at maximum). It does, however, experience varying degrees of QoS degradation. The ultimate application QoS cannot be restricted by ASAR-only recovery. We provide a hybrid recovery (ASAR + Rerun) approach to solve this problem and examine the performance-quality trade-off curve at a finer-grained level. In this part, we use QoSmod to assess the efficacy of hybrid recovery. Execution speed and QoS level for all six apps using hybrid recovery. The same units are used on all axes, each subgraph's top horizontal line reflects the flawless QoS attained by error-free or rerun-based execution. . The user-specified QoS threshold (QoSthd) is displayed in the second horizontal. We evaluate our QoSmod for significantly more precise undetected failure rates (1%—50%). The greater failure rate investigates near-threshold operations that are more forceful.

The hybrid recovery process chooses an approximation ratio while attempting to conservatively match a described point in QoSmod. The findings indicate that hybrid. Discard Recovery is a method for recovering from unsuccessful Try blocks that evaluates the discard mechanism. The six apps' QoS and execution times while employing the discard method The ideal QoS is shown by the two horizontal lines at the top of each subgraph employing rerun-based recovery and a user-specified QoS threshold (QoSthd). The discard

mechanism offers the biggest performance gain with an execution time that is between 30% and 70% quicker than ASAR. Yet the output's observed QoS (Discard) is below the QoS threshold (QoSthd). Cutoff failure rate is the Try block failure rate at which the QoS falls below QoSthd (fc). Certain applications, like MC and A2T, have fc values as low as 1%. Programs like MonteCarlo, for instance, do not enable discard-based recovery, because deleting a sub-computation prevents the algorithm from computing the final result, leading in a 100% QoS loss. The discard mechanism has a QoS loss that ranges from 16% to 100% overall. These limitations prevent the discard technique from being used if the user demands exact output quality assurances.

Program refactoring is made possible by the ASAR, or application-specific approximate recovery, technique that is proposed. . Whereas non-critical code allows the programmer to provide application-specific flexibility, critical code is intended to function exactly as conventional software does. They can be utilised collectively in an approximation of a computer system model. By employing a rough substitute, this reduces the cost of software-based mistake recovery in comparison to rerunning. We provide a hybrid recovery strategy to ensure output acceptability and investigate the performance-quality curve at a finer granularity. In hybrid recovery, a variety of software-based recovery techniques are combined with a well-defined QoS paradigm. We implement an instance of hybrid recovery utilising ASAR and rerun recovery method. . We demonstrate that the proposed hybrid recovery may work at any intermediate position on the performance-quality trade-off curve for test inputs by characterising a QoS model using training inputs. Our findings show that hybrid recover assures output quality above the user-specified threshold and offers 1.5%–11.6% quicker execution time than the rerun approach. Moreover, we demonstrate that the discard method executes on average 30%–70% quicker than ASAR, but may result in an unacceptable QoS deterioration.

CHAPTER 3

RELIABILITY & PERFORMANCE TRADE-OFF STUDY OF HETEROGENEOUS MEMORIES

Vanitha K, Assistant Professor,
Department of Computer Science and Engineering,
Jain (Deemed to be University) Bangalore, Karnataka, India
Email Id- k.vanitha@jainuniversity.ac.in

The computer architects have concentrated their efforts on heterogeneous memories, arranged as die-stacked in-package and off-package memory, to increase memory bandwidth and capacity. Researchers have investigated strategies and structures to improve functionality by speeding up access to quicker die-stacked memory. However, the lack of thorough research on the dependability of such arrangements makes them less desirable for data centres and mission-critical systems. Field experiments reveal that error correction codes and device physics affect memory dependability (ECC). The performance-critical in-package memory may prefer weaker ECC systems than off-chip due to the capacity, latency, and energy costs of ECC. These systems are further enhanced to operate at maximum efficiency by speeding up access to high-performance in-package memory. In this chapter, writers do a trade-off analysis on the performance and reliability of heterogeneous memory architectures using real-world DRAM failure data (HMA). The issue that an HMA system that merely optimises for performance may have degraded dependability over time is presented in this chapter. An age-aware access rate control technique is also suggested in this chapter to guarantee the dependable functioning of legacy systems.

The development of memory technology

We can enhance capacity and achieve high bandwidth by die-stacking DRAM in a single package. Several ideas combine in-package (high bandwidth) memory with off-package (high capacity) memory in a single system. The highest-performing memory device should get as many accesses as feasible for the system to function at its best. The majority of this study focuses on tracking, predicting, and migrating data to the best place in an effort to improve performance or use less energy. While capacity and bandwidth have increased, the reliability and serviceability of heterogeneous memory still face significant difficulties. Error prevention is provided by off-package DDR DRAM utilising methods like ChipKill. Although powerful ECC algorithms have been suggested for in-package memory, the limited capacity of these memories suggests that lower-overhead ECC is more likely to be used. For instance, off-package memory employs extra DRAM chips to offer ChipKill, but for in-package die-stacked memories, adding more DRAM dies is costly. For the sake of this study, we assume that in-package die-stacked memory employs single error correct-double error detect (SEC-DED) ECC and that off-package DRAM employs single-ChipKill ECC. We shall simply refer to single-ChipKill as ChipKill for the remainder of the thesis.

Memory error types

Mistakes may result from either temporary or enduring problems. Scrubbing is a method that may, to a certain degree, rectify the accumulation of transient errors. Memory scrubbing lessens the possibility that transitory defects cause error accumulation that is greater than what error-correcting codes can handle. The memory controller examines every memory

location during scrubbing and fixes any mistakes. Permanent fault errors cannot be fixed by scrubbing and may eventually cause a fault accumulation. The memory system becomes increasingly vulnerable as its ages to an unfixable (and ultimately undetected) malfunction as a result of the accumulation of these defects. An unfixable memory issue may cause a computer to crash, damage application data, and introduce security vulnerabilities. Thus, dependability is crucial to prevent system outages and guarantee proper and secure functioning. Bad DIMMs in off-package memory may be changed to preserve dependability when the memory is becoming older. Nevertheless, to replace the die-stacked in-package memory, a new chip is needed.

A research of DRAM failures in the field was published by Sridharan et al. in 2012. Their research demonstrates that, compared to SEC-DED, ChipKill significantly lowers the node failure rate attributable to uncorrected DRAM faults. As a result, compared to accessing a more reliable off-package memory, each access to a less reliable, die-stacked main memory device reduces the system dependability. In this chapter, we demonstrate how using in- and off-package memory wisely may lower the failure probability by 26x compared to a system that only uses the fastest in-package memory for all memory requests. We provide an analysis of the reliability-performance trade-off in HMA systems. We demonstrate that compared to the failure probability in year 1, the system failure probability might rise by more than a factor of 1000 in the seventh year. We demonstrate that low and medium bandwidth benchmarks may achieve higher reliability (or lower failure probability) by using just off-package DDR memory at the expense of 37% and 45%, respectively, of performance deterioration. When all of a high bandwidth benchmark's memory access is limited to more dependable off-package DDR memory, performance falls by 51%.

Differentiated Memory Architecture

Several memory modules make up a heterogeneous memory architecture (HMA), such as an HMA system having on-package die-stacked DRAM and off-package traditional DDRx memory. The dependability issues that die-stacked memory faces in comparison to traditional DDRx memory will next be covered. The enormous bandwidth and capacity demands of modern applications operating on highly parallel multi-core and GPU systems gave rise to HMA. This encouraged hardware designers to combine traditional off-package DDRx with more recent 3D high-bandwidth memory. Although conventional DDRx memory reliability has been studied for decades by both industry and academics, 3D die-stacked memory reliability research is still in its infancy. As a result, these memory systems often provide heterogeneity in their dependability. Off-package memory is shown as a 2D arrangement of DRAM chips, each of which offers a set amount of bits, 8 bits (8x arrangement), and every cycle. One cache line is provided by a group of DRAM chips that operate in lockstep (64-bits). Single-bit correction and double-bit detection are provided via an extra 8-bit chip (ECC). Moreover, with a 4x DRAM configuration, we may provide greater ChipKill symbol-based correction, which necessitates dispersing ECC bits among many DRAM chips. Die-stacked memory uses a single DRAM chip to provide the full cache line, resulting in increased bandwidth and a single DRAM chip's capacity of 128 bits per cycle. Thus, it is impossible to directly apply ChipKill-based Reliability, Availability and Serviceability (RAS) in die-stacked DRAM due to its fundamental nature. A performance overhead develops as a consequence of the reduction in bank-level parallelism caused by dividing a cache line among many DRAM chips. Moreover, the power/energy overhead of die-stacked memory is increased by activating many chips. Sim et al. describe a costly modification to the traditional ECC-based architecture for die-stacked DRAM known as ChipKill-level RAS. With its 3D die-stacked memory, Micron's Hybrid Memory Cube (HMC) offers exceptional stability. Yet it need unique DRAM components and memory controller modifications. Hence, the

performance, energy, and complexity of each of these options must be considered. The field's research has produced a number of creative, low-cost methods to improve the dependability of die-stacked memories. The same is true, however, for traditional off-package memory. So, it is reasonable to infer that the trustworthiness of the various memories in an HMA system will continue to vary.

Simulated reliability tests: Data on DRAM failure rates are taken from and used in the simulation framework. FaultSim is a quick and precise tool for our reliability studies because of its event-based design and real-world failure data. We conduct the specified number of simulations. Based on their FIT (Failure in Time) rates, each simulation involves injecting a defect into a bit, word, column, row, or bank. The chosen error-correction algorithm is then implemented, and the result is reported as a discovered, corrected, or uncorrected mistake. When the failing bit is utilised by the programme, an error occurs. Injected faults might be temporary or permanent. FaultSim makes the cautious assumption that every DRAM malfunction causes an error, which gives our research an upper limit on the error rate. The flaws are included to imitate a 348-week period of time (approximately 7 years). To calculate the failure risk of a heterogeneous memory architecture, we utilise the likelihood of uncorrected mistakes. The SEC-DED and ChipKill error codes are simulated by FaultSim. The initial dependability and ageing curves of SEC-DED and ChipKill are dissimilar. Using level one (M1) memory equipped with SEC-DED and level two (M2) memory with ChipKill, we mimic a heterogeneous memory architecture. SEC-DED offers less complexity and power, whereas ChipKill offers more dependability. In order to decrease latency and capacity overhead, in-package die-stacked memory may provide worse reliability when employing parity-based detection algorithms. Hence it is reasonable to assume SEC-DED for in-package memory. Also, the eight times greater density of in-package die-stacked memory and emerging failure mechanisms such defective through-silicon vias (TSV) increase the likelihood of failure. scales FIT rates by eight times of the real-world DRAM field study and conducts a sensitivity research by sweeping FIT rate for TSV failures in the absence of field study data on failure rates of die-stacked memory. We highlight the issue caused by the varying dependability levels of in-package and off-package memory in this paper. The difference in the dependability levels will only widen and strengthen our case if we scale the FIT rate of die-stacked memory by eight times and take into account newer failure types. To be as cautious as feasible for our investigation, we do not scale the FIT rates for die-stacked DRAM. Hence, with the same densities and FIT rates, we replicate memory M1 and M2.

Simulations of performance

Performance is increased at the expense of dependability when more frequently visited pages are stored in package memory. We employ the off-package and in-package die-stacked DDR3 and HBM memory standards, respectively, for performance assessment. Similar latencies of 40ns and 45ns are shared by HBM and DDR3 memory. Yet, depending on the organisation, HBM offers 2X–8X more bandwidth than DDR memory. So, depending on the workload, utilising one memory over another might affect performance. We go through our performance assessment process using Ramulator in this portion. The DRAM simulator Ramulator offers cycle-accurate performance simulations for many memory standards, including DDR3/4, LPDDR3/4, GDDR5, and HBM. The simulator accepts an input trace file while it is in trace-driven mode. The number of non-memory instructions that were executed before each memory request, the memory location, and the request type (read or write) are all included in the trace file. Ramulator can only imitate one level of memory at a time. To mimic two layers of heterogeneous memory, we extended Ramulator. Using HBM as in-package memory and DDR3 as off-package memory, we simulate a 16-core machine. Table 1 contains the entire Processor and memory configurations. used to produce memory traces,

while Moola is used to filter cache. As indicated in Table, we employ three workloads from the SPEC benchmark suite. We picked the SPEC benchmarks mcf, astar, and cactus, which correspond to high, medium, and low MPKI, respectively. We organised all of the SPEC benchmarks in increasing order of their MPKI (Misses Per Kilo Instructions) values. To change the access rate to HBM, we fixed a set of randomly chosen pages in HBM memory and the other pages in off-package DDR3 memory. To accommodate Astar and Cactus' whole working set, 4GB of HBM memory is emulated. Workloads into 16GB (HBM) for mcf and in-package RAM. As level 2 off-package memory, we utilised 16GB of DDR3 memory.

Performance vs. Reliability Trade-Off

This study's objective is to assess the trade-off between performance and reliability. The left y-axis represents normalised performance, while the right y-axis represents the failure probability in weeks 0, 180, and 348. Week 0's failure probability is little impacted by growing access rate. Yet, increasing the access rate will have a negative impact on the failure probability as the system matures and develops permanent memory defects. The dotted line in the illustration depicts the performance ceiling of 80% IPC. Running a task at 80% of its highest IPC reduces failure probability on average by 5.15 104, 72.29 104, and 137.98 104 in weeks 0 through 348, respectively.

Access Rate Control for seniors

This section presents an aging-aware access rate management method and expands on the performance and reliability trade-off. In simulations with a limit on the failure chance, we tested how high, medium, and low bandwidth benchmarks performed throughout the course of the system's lifespan. The simulations' outcomes the access rate control mechanism kicks in whenever the failure probability is 300x greater than the original failure probability, according to simulations we did with a maximum of 300x on worsening of the failure probability. Up until the failure probability ceiling is met, the system may operate at maximum efficiency. When the system passes the failure probability cap, T1, that week is noted. Depending on the access rate for their peak performance, various workloads achieve T1 at different times over different weeks. For instance, although mcf could continue operating until week 132 without exceeding the failure probability barrier, cactus exceeds the failure probability cap in week 108. Access rate is lowered every week by a defined amount when the failure probability limit is reached by selecting a statically profiled random page placement.

FIT DRAM rates:

In the field research work, trials are carried out on 2.69 million DRAM devices, and average FIT Rates are given over a period of 11 months for different DRAM components (bit, row, column, word, bank, rank), for both transient and chronic failures. We calculate the failure probability over a 7-year period using these average FIT rates. As a result, we account for FIT rate variance among various DRAM components in our simulation. While it is an assumption in our model, assuming a constant FIT rate throughout the course of seven years As far as we are aware, there is no field research information on die-stacked memory error rates. Large-scale research such to the DRAM field study will become possible when heterogeneous memory permeates commercial data centres, and studies like ours may integrate actual defect and error rates. It will become possible to assess their impact by improving fault simulators with additional protection techniques as the possibilities for protecting in-package memory expand.

Environment-related changes to FIT rates

Geographical characteristics may have a considerable influence on FIT rates, according to

field study. A system positioned at an altitude of 7,320 feet is more prone to transitory mistakes than a system at an altitude of 43 feet, according to the extended field research work. In order to reveal a security hole, advantage of the increased sensitivity of DRAM rows to voltage variations. As a result, in the presence of the differences mentioned above, our algorithm's prediction of T1 in the field will need hardware and software adjustments. Machine-check architecture (MCA) registers are provided by the x86 architecture and may be used to record rectified error occurrences. The information from MCA registers should be frequently logged into a log file by the operating system. The log files may include a range of data, including the memory locations where the errors are occurring, the time when they occurred, the kind of error (corrected or uncorrected), and ECC syndrome bits. The aging-aware access rate control module may regularly review the log file to determine when to begin lowering the access rate to in-package die-stacked memory. To count the amount of rectified and uncorrected mistakes in different DRAM components, employed similar HW/SW facilities in their field investigation of DRAM faults.

Expense of recoverable mistake and uncorrectable error

The trade-offs between performance and the likelihood that an unfixable issue may fail. System downtime, component replacements, data loss, or security vulnerabilities are the costs of an unfixable memory mistake. Error-correcting codes are thus used in memory systems, such as those found in Google data centres, to recover from such faults. Yet, there is a cost involved in fixing a mistake. Performance, area, power, and dollar cost are the four main divisions of the cost. Every time a memory access is made, error detection must be activated in order to recover from a memory error, for example, ECC must read an extra 8 bytes of ECC-bits for every 64 bytes. Due to the possibility of parallel execution, error detection has a little performance penalty. A 12.5% area, bandwidth, and power overhead does exist, however. The additional cost of employing an ECC DIMM for DDR memory is around 10% to 20% more than that of a non-ECC DIMM. In comparison to off-package DDR memory, the cost of allowing recovery for 3D die-stacked memory is greater.

It has assessed how well the present heterogeneous memory architectures trade off performance and reliability. We go through the variations in dependability characteristics between off-package high capacity DDR memory and die-stacked high bandwidth memory. We also talk about the architectural difficulties in improving die-stacked memory dependability. We contend that system designers should take aging-aware data placement strategies into consideration in the face of architectural obstacles to improve the dependability of the present die-stacked memory. During the course of the system's lifespan, the dependability will decline with a data placement strategy that simply prioritises performance. We suggest and demonstrate the advantages of a policy that utilises access rate as a control to lower the likelihood of system failure to 137.98 x 104 in the last week, which is a 26x decrease from the first week. In order to assure decreased failure probability, we also provide an aging-aware access rate management strategy that reduces access rate to the die-stacked memory with poorer reliability as it ages. As a consequence, switching from a performance-focused to an aging-aware use of HMA will lead to a prolonged and trustworthy functioning in the system's last years.

Developing Memory Units and Temporary Faults

The dependability of computer systems is seriously threatened by transient problems resulting from single event upsets (SEUs). The vulnerability of semiconductor devices to SEUs has risen as a result of technology scaling and decreased operating voltages. These errors may cause erroneous bit flips that damage the architectural state, causing random crashes, data loss, and security flaw. Die-stacked and standard DDRx memory exhibit notable

variances in their dependability, as stated in the preceding chapter, in addition to disparities in latency and throughput. Due to greater bit density and novel failure modes (TSV failure), die-stacked memory has a higher fault rate. Due to cost and complexity restrictions, die-stacked memory often uses inferior error correction. Worse overall dependability is the outcome of a naïve performance-focused data placement method that inserts frequently visited memory pages (hot pages) in die-stacked memory. Several placement techniques that place more or less of the most frequently visited pages in stacked DRAM; we can see that the reliability loss required to get maximum performance may be rather severe. In this chapter, we examine the hotness and susceptibility of memory pages in order to create a reliable data placement strategy for the HMA system.

We employ Architectural Vulnerability Factor (AVF) analysis to measure the susceptibility of memory pages. The likelihood that a transient defect may cause an observable programme error is known as the AVF of a memory page. So, a memory page with a higher AVF (high-risk) is more likely than one with a lower AVF to cause improper programme execution (low-risk). This study's major finding is that page hotness (access rate) and AVF are not always connected with one another. In fact, we demonstrate that programmes may include up to 39% of hot and low-risk memory pages. This attribute opens up the option of creating placement schemes on a hybrid memory architecture that maintain both reliability and performance; specifically, we can work in the upper right area, which is unavailable to placements that are just performance-focused. According to our findings, a static placement may minimize mistakes by 2X at a 1% performance cost, whereas a dynamic strategy can increase dependability by 1.9X at a 13% performance cost. The contributions made by this chapter are as follows:

application data are quantitatively divided into low-risk and high-risk groups. Our findings indicate that low-risk hot data makes up between 9% and 39% of the overall memory footprint of apps. an algorithm that enables inexpensive dynamic risk monitors for data in memory. Use static, heuristic-based, and dynamic migration-based strategies to illustrate the possibilities of reliability-aware data placement for heterogeneous memory architectures (HMAs). To offer a lightweight dynamic reliability-aware migration method for HMAs, we cross breed two distinct counters and propose and analyse Cross Counters.

Definitions and Terminology

A few crucial dependability definitions, formulae, and equations that are crucial to this study are reviewed. Also, we go into the history of heterogeneous memory architecture (HMA). To illustrate our methods, we will use the reliability and performance heterogeneity shown by HMA systems. The techniques created in this study, however, may be used to any memory architecture that provides this level of variety. Systems that include non-volatile memory or other cutting-edge memory technologies into the hierarchy, for instance, would exhibit more heterogeneity along both axes.

Mistake in Time (FIT Rates)

The raw failure rate of a device owing to single event upsets (SEUs) or the failure in time (FIT Rate) of a device are both dependent on the circuit's properties and the neutron flux in the surrounding area. The cross-sectional area of the circuit, the charge needed to toggle a bit (create a defect), and the charge collecting efficiency all have an impact on FIT rates. The cross-section area and charge required to toggle a bit are less because to shrinking technologies. The number of bits per unit area rises with each new process technology, exponentially causing a rise in raw FIT rates owing to SEUs.

Building Vulnerability Factor (AVF)

An observable programme error will most likely come from a transitory malfunction in a hardware structure, according to the definition of its AVF. Mukherjee et al. provide a method to calculate a processor's AVF. The bits of a hardware structure that the authors watch are divided into two categories: (a) those required for architecturally correct execution (ACE bits), and (b) the remaining bits, often known as un-ACE bits. In the absence of error correcting methods, a failure in an ACE bit will cause a programme observable error, but a fault in an un-ACE bit will not have an impact on the program's output. Since a defect in a predictor bit only impacts performance and not programme output, all branch predictor bits, for instance, are un-ACE. A bit may only be ACE for a portion of the time the programme is running and un-ACE for the remainder. For instance, a physical register (R1) that is written at the start of the execution, read halfway through, and then dead thereafter is only in the ACE state for 50% of the whole execution duration. The percentage of the overall execution time that R1 spends in the ACE state is its AVF. It expresses the AVF of a hardware structure Mi with bit size BMi over an N-cycle time.

Performance-focused Static Data Insertion

We investigate a profile-guided static page placement to get the top limit on performance for HMA systems. To start, we profile each task to learn about the hot pages. Next, depending on the number of accesses, we choose the top 1GB of hot pages and move them to high-bandwidth low-reliability memory while moving the remainder pages to high-reliability DDRx memory. The outcomes of static placements for all workloads that are performance-focused. When HMA is placed with performance in mind as opposed to merely using DDRx RAM, we see an improvement in performance across the board for all workloads (IPC on the left y-axis). Applications operate 1.55 times faster than when using just DDRx memory, on average. Yet we also see a sharp rise in the soft mistake rate (SER on right y-axis). We notice 266x boost in SER on our test suite vs to simply DDRx memory. Hence, storing hot pages in stacked memory dramatically exposes the system to the less dependable memory. If hotness and AVF are highly correlated, as these results seem to indicate, then we are forced to make difficult trade-offs between reliability and performance. The next section examines this correlation more closely.

AVF vs. Hotness of a Memory Page

To understand the relationship between page hotness and vulnerability (risk) we simulate the single memory architecture using only high-reliability DDRx memory. We place the entire memory footprint of a workload in DDRx memory and run the simulation using our modified Ramulator. the top 1000 hottest pages of a workload (mix1) arranged in decreasing order of their hotness. The left y-axis represents the page hotness measured using accesses to a page and the right y-axis represents the page vulnerability measured as AVF percentage. The graph shows that most of the hot pages have AVF at around 80%. However, there are pages (in the top 1000 hot pages) which have AVF below 60% and as low as 5%. Hence, we conclude from that page hotness and AVF have a weak correlation. The correlation coefficient between hotness and AVF for the entire memory footprint is a very low 0.08. Thus, there exists an opportunity of identifying hot page with low-AVF (low-risk) (low-risk).

In order to quantify this opportunity, we divide the entire memory footprint into hot & cold pages and high- & low-AVF pages. We split the entire memory footprint of each workload around mean hotness and mean avf values as shown by the scatter plots. Each memory page of the workload is plotted as a point on the scatter with its AVF along the x-axis and hotness along y-axis. The horizontal line represents mean hotness (access count) and the vertical line divides the memory footprint into low-AVF (low-risk) and high-AVF (risk) pages. These two lines divide the entire memory footprint into four sections: (i) hot & high-AVF, (ii) hot &

low-AVF, (iii) cold & high-AVF, (iv) cold & low-AVF. We observe the presence of pages in all four sections in all workloads, although lbm is an outlier with few in the upper left. Generally, we find a considerable number of pages in the upper left quadrant (hot & low-AVF) (hot & low-AVF). For the mix1 workload we find 1.66GB of pages which qualify as hot & low-AVF pages. Such pages are ideal candidates for high-bandwidth low-reliability memory to allow performance-focused reliability-aware operation. Different workloads have a varying number of hot & low-AVF pages. For workloads under this study we find that hot & low-AVF pages ranges from 9% to 39% of the entire memory footprint.

Reliability-aware Static Data Insertion

We explore the use of data vulnerability (AVF) and hotness (access count) in- formation to find a placement for HMAs. An ideal data placement for an HMA system operates near the IPC of performance-focused placement and near the SER of only DDRx memory. Thus, the goal of this section is to achieve high IPC with SER well below that of die-stacked memory.

AVF-focused Static Data Placement

Similar to the performance-focused data placement technique, a naive reliability- only technique places the least risky pages (i.e. the least AVF pages) in low-reliability memory and the rest in high-reliability memory. We use the AVF profiling information similar to the profile-guided performance-focused data placement to find and place pages with the least AVF in low-reliability memory. The lowest AVF pages start from the left side. An AVF-focused placement doesn’t take hotness into account and it may end up picking pages from both the top left and bottom left quadrants. We refer to this placement as AVF-focused data placement for HMAs as the performance (IPC on right x-axis) and reliability (SER on right y-axis) are normalised to IPC and SER of only DDRx memory, respectively. The stacked bars show the performance loss compared to performance-focused placement. On average we observe that the soft error error rate reduces to 48x from 266x relative to the SER of only DDRx memory. However, AVF-focused data placement suffers from 19% loss in performance relative to performance-focused data placement.

Different workloads may choose cold pages for high-bandwidth memory, but the performance hit will depend on the bandwidth needed and the hotness spread between pages (scatter plots). arranged according to the MPKI in decreasing order (Misses Per Kilo Instructions). The workloads on the right side of the graph are latency-sensitive, whereas the workloads on the left are bandwidth-intensive. Performance of bandwidth-intensive workloads is more impacted by moving hot pages from high-bandwidth memory to off-package DDRx memory than it is by latency-sensitive workloads. This explains why workloads other than lbm and milc suffer from significant performance loss. With AVF-focused data placement, the loss for lbm and milc is just 6% and 1%, respectively. The access counts for lbm and milc are very consistent across pages. So, for lbm and milc, it doesn't really matter which pages are transferred to high-bandwidth memory in terms of speed. AVF-focused data placement, however, often runs the risk of choosing cold pages, which lowers performance. Hence, decreasing the soft error rate by employing solely AVF information is possible, but at a large performance penalty.

AVF-aware Static Data Placement with Performance Focus

We investigate performance-focused AVF-aware data placement to assure both performance and reliability. We choose hot, low-AVF pages and store them in memory that has a high bandwidth but poor dependability. The dependability (SER on the right y-axis) and performance (IPC on the right x-axis) were standardised to the SER and IPC of DDRx memory alone, respectively. The stacking bar depicts the reduced performance as opposed to

static placement that is performance-focused. We see that, when compared to simply DDRx memory, the average SER is still extremely near to the SER of AVF-focused placement, at 48x vs. 57x. Yet, there is a 10% performance loss on average when compared to static placement that is performance-focused. While precise AVF estimate or prediction is a challenging topic, this static strategy depends on oracle knowledge of AVF. In that chapter, Walcott et al. estimate the AVF of the instruction queue, load-store queue, and register files using IPC as a heuristic. They show that lower AVF is caused by greater IPC, but the heuristic is useless for analysing specific memory pages.

Memory Page Heuristic-based Static Data Placement Write Ratio & AVF: A memory page with a higher ratio of writes to reads is likely to have a lower AVF than one with a higher ratio of reads. Remember that most "dead" data periods finish with a write, therefore a higher number of writes indicates a higher number of dead intervals. The chance of extended dead periods is increased by a high ratio of writes to reads. The top 1000 hot pages for the mix1 workload a along with the write ratio (Wr/Rd ratio) on the left y-axis and AVF on the right y-axis. The relationship between AVF and write ratio is much more correlated than it is with hotness. We find a -0.32 negative connection between write ratios and AVF, which, although not very strong, gives us some possibility to quickly get an estimate of AVF. Utilizing the Write Ratio to Calculate the AVF & Hotness

It demonstrates the effectiveness of the Wr ratio and Wr2 ratio data placement approaches. Wr2/Rd is used to measure the latter. To execute a) and b), we look for sites with high Wr and Wr2 ratios. The remaining pages go into high-reliability memory, whereas these pages are stored in low-reliability memory.

Migrations that are reliability-aware

We can now determine the hotness of a page (reads + writes) and the risk ratio, which is defined as Wr/Rd, by simply dividing the set of counters into two sets, one for reads and one for writes. We do not have to compute Wr2/Rd because we have a precise way to quantify page hotness. To categorise memory pages visited during the period into hot and cold and high-risk and low-risk, respectively, we utilise mean hotness and risk values as criteria. Our technique makes an effort to swap out all cold and high-risk pages that are presently stored in low-reliability memory with hot and low-risk pages from high-reliability memory at each interval. This reliability-conscious migrations technique is also known as the Full Counter-based (FC-based) mechanism. Reliability-aware migrations employing Full Counters (FC) provide the IPC (left y-axis) and SER (right y-axis) findings; both values are normalised to that of just DDRx memory. The stacking bar depicts the performance degradation when performance-focused migrations are employed. We see varied degrees of performance degradation across various workloads. Due to fewer migrations than with perf-focused placement, we saw a modest speedup in milc workload. In comparison to performance-focused migrations, we often see a drop in SER with FC-based migrations, from 239x to 120x at a 13% performance loss. As a result, we are able to attain a degree of dependability comparable to static heuristic-based reliability while employing heuristics-based runtime risk estimate.

Cost of dynamic schemes on hardware

The hardware cost of the dynamic reliability-aware migration strategies using full counters that were previously provided (FC). For the purpose of detecting migration candidates and accurately retrieving the migrated pages, all dynamic memory managers need accounting structures (typically a remap table). For each page in memory, the FC-based reliability-aware migration method keeps two sets of counters to track the number of reads and writes. Our

investigation shows that the greatest values we saw can be stored in 6-bit counters. For the most part, we presume that such a system is built using 8-bit counters that can accommodate up to four times greater numbers. In order for the counters to avoid overflowing following access to a page with the maximum count, we also assume that they are saturating. According to our presumptions, this mechanism's activity monitoring (identifying migration candidates) component needs 16 bits per 4K page. With our example 17GB HMA (1 GB HBM and 16GB DDRx), we have a total of 4.25M pages, thus 8.5MB will be required to hold the Whole Counter data. In addition to activity monitoring, an OS-transparent technique needs a remap table that can accurately retrieve migrated pages, maybe numerous times. When the same page is remigrated, a naively created remap table (page ID Remap address) fails. We want an enhanced remap table with two entries per page ID, one for the remapped address and one for the page ID that is presently in use, in order to implement the re-migration capability. In our example heterogeneous memory architecture, the overall cost for this remap table is 36MB.

Activity Tracking Optimisation

It is challenging to implement the suggested reliability-aware technique in reality due to the size of bookkeeping structures. In addition to the memory need, sorting the sets of counters at per interval to find migration candidates is expensive. Every time a memory request comes in, we need to query our remap table. Recent research has shown that intelligently caching a small portion of the remap table while keeping the remainder of the table in main memory may minimise the amount of memory visits and lookup-related performance overhead. There are other methods as well, such keeping a table that may be arbitrarily small, and when it fills up, the OS can step in and update the TLB and page tables. The Meswani et al. technique directly calls the OS and does not need a remap table. We concentrate on reducing the HW cost of activity tracking in light of existing options to minimise the HW cost of remap tables.

Migrations in Motion Using Cross Counters (CC)

Using a newly suggested method based on the Majority Element Algorithm, we can considerably reduce the hardware cost of activity monitoring devices. With only a few MEA counters, a memory management system can effectively estimate the set of the most popular pages throughout a period. Due to its innate preference for temporal locality, MEA has been shown to be more effective than the far more costly method of maintaining complete counters. It may be explained as two unique units maximising two various features. In order to boost speed, one of the devices swaps out cold pages from high-bandwidth memory for hot pages from slower DDRx memory. The second unit, on the other hand, aims to increase reliability by swapping out high-risk pages that are present in the low-reliability memory with low-risk pages from the high-reliability memory.

We suggest the "Cross Counters" reliability-aware migration strategy in light of this insight. MEA and Full Counters are combined in Cross Counters (CC). Our performance-optimizing unit uses MEA. We discover up to 32 globally hot pages in each interval using just a 32-entry MEA map. Cross Counters' reliability-optimized unit still employs a complete counter strategy, but only for pages in low-reliability memory. These counters maintain individual read and write counts and function as specified in the reliability-aware dynamic method before it.

CHAPTER 4

IMPLEMENTATION OF CROSS COUNTERS

Sonal Sharma, Associate Professor,
Department of Computer Science and Engineering,
Jain (Deemed to be University) Bangalore, Karnataka, India
Email Id- s.sonal@jainuniversity.ac.in

MEA-based migration algorithms perform best at very short intervals because they can often push a tiny number of pages to high-bandwidth memory for verification, we simulated and contrasted Cross Counters with MEA intervals set at 50 and 100 s. Our findings support their estimates. This period is known as the MEA-interval. Moreover, FC-based migrations need substantially bigger intervals since they try to migrate a sizable number of pages at once, necessitating the use of extremely large intervals for FC-based migration techniques. As we are utilising a dynamic Wr ratio, a significant amount of reads and writes must be gathered in order to provide a more precise risk assessment. We configured the two management units in Cross Counters to run at various intervals. Every 50 s, the performance unit (MEA) migrates a small number of pages to the fast memory (MEA-interval). The interval size of the dependability unit is configured to be 200x greater (FC-interval).

The last feature is that the performance-focused unit orchestrates migrations in Cross Counters (CC). Every time it finds a candidate, this unit will shift pages into fast memory. To determine if a high-risk page is awaiting transfer to the high-reliability memory, CC will first contact the reliability unit. In the event that one does, it immediately becomes the second migration candidate, and migration starts.

The performance unit will go on and replace another page in high-bandwidth low-reliability memory that is anticipated to be cold if there are no outstanding high-risk pages (for example, within the first 200 intervals of execution). Reliability unit doesn't really cause any migrations at FC-interval; it just sorts and refreshes its counts. The findings of IPC (left y-axis) and SER (right y-axis) utilising reliability-aware data placement based on CC are normalised to IPC and SER of solely DDRx memory. In comparison to dynamic performance-focused migrations, the soft error rate is decreased from 239 to 140 times for a 15% performance loss. Cross Counters raise the soft error rate to 140x from 118x when compared to dynamic reliability-aware migrations using Full Counters (See Table 5.2). By enabling hot and "possibly" dangerous pages to transfer from high-reliability memory to high-bandwidth memory at MEA-interval, the soft error rate has increased. A page is moved back to high-reliability memory at the FC-interval border if it becomes unsafe while in the FC-interval. Keep in mind that the page remains in high-reliability memory if it doesn't become heated again. In contrast to pure history-based FC-based reliability-aware migrations, workloads like astar that have the majority of their pages hot during execution will face an increase in the number of migrations and a corresponding decrease in performance.

We saw a large performance boost in workloads with striding patterns as a result of the introduction of MEA counters that account for the temporal characteristics of an access pattern. For instance, when comparing the cacti ADM workload to just history-based FC-based migrations, we saw an 11% performance improvement. The next section goes over how, on average, CC puts the soft error rate very near to FC at a very cheap hardware cost.

HW Cost for Cross Counters

The Cross-Counters method dramatically lowers the overhead associated with activity monitoring. MEA counters can profile the whole memory space and pinpoint anticipated hot pages with only 100KB. Due to the fact that we only employ FC for risk monitoring in low-reliability memory, the necessary cost rises to 512kB. The CCs technique lowers the overall cost of activity monitoring to 612kB (about 14 times less than global complete counters). Other advantages come with reduced memory requirements: As a result, (a) it is now simpler to keep all counters on a chip without the requirement for caching, and (b) the overhead of each interval is greatly reduced since we only need to sort 250 000 counters rather than 8.5 M.

Mitigation of Hardware-based Errors

Techniques including redundancy, design-time tweaks, and architecture-level methods are used in hardware-based error mitigation. Hardware logic duplication is used by Triple Modular Redundancy (TMR) and Double Module Redundancy to identify and fix problems. Commercial systems have made use of hardware error mitigation strategies based on redundancy. Several hardware-based memory protection strategies have been developed, including parity and error correction codes, with strategies to boost their effectiveness and lessen performance penalty (XED).

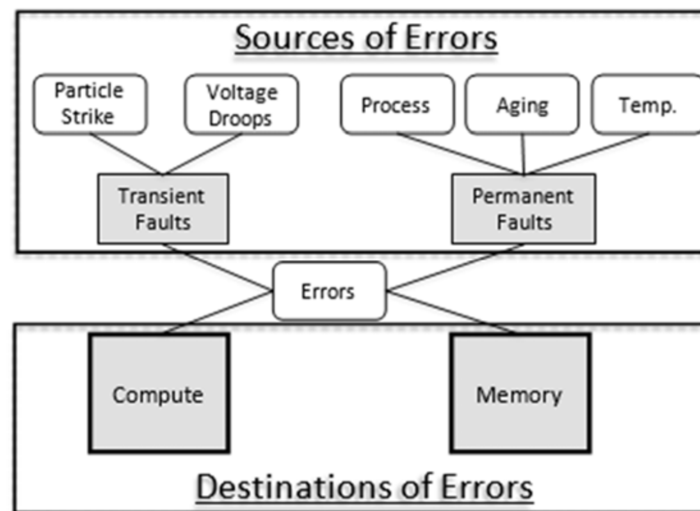

Figure 4.1. Illustrates the software-based error mitigation techniques for both compute and memory units.

The development of multi-core CPUs and GPUs brought: Redundant Multithreading (RMT) to the attention of the scientific community. Software-based fault detection support for CPUs utilising process-level redundancy is what Shye et al (PLR). PLR utilises dynamic binary instrumentation to inject the synchronisation code when replicating a programme at the process level. As system calls depart PLR's SoR, PLR incurs overheads from dynamic instrumentation and barrier synchronisations every time a system call is made. Using speculative execution, RAFT, which lowers the synchronisation cost of PLR on a multicore Processor. The system call of the first thread to reach the synchronisation point is hypothetically executed by RAFT. It is not feasible for the present GPU designs to control GPU threads at the process level. As our design uses ECC support to protect memory, PLR and RAFT do not need complete duplication to preserve memory structures. Several hardware and software approaches to CPU RMT have also been put forward. a hardware-

based error detection approach using fingerprinting for CPUs. For each instruction that executes in lockstep, the task compares the complete CPU state, including register values, store values, and effective addresses. By modelling workloads and corresponding fingerprinting hardware using Simple Scalar, they show the decrease in communication bandwidth in their work. We put fingerprinting into practise for Inter-Group RMT, where threads don't run in lockstep, and demonstrate how our method cuts down on both locking overhead and communication bandwidth. On actual hardware, we fully implement fingerprinting as a section of a compiler transformation for GPUs (Figure 4.1).

A preliminary investigation into GPU-based software-based RMT. Their methods are hand-coded and based on older VLIW architectures, which restricts their usefulness. A totally software-based method for GPU RMT was disclosed. The Wadden et al. core RMT algorithms were transferred by Charu et al. to an HSA compiler, and their work was assessed on a GCN2 APU device that complied with HSA. To create Intra- and Inter-Group GPU RMT, Wadden et al. employed a discrete GPU and an OpenCL programming environment. A swizzle instruction may be used, according to Wadden et al. They customised the shader compiler (compiler backend) to meet their architecture and demonstrated how Intra-Group RMT may benefit from hardware improvements. For certain benchmarks, their method has a performance reduction as a result of casting and packing/unpacking of communicated data via 32-bit registers. The portable compiler transformation pass that we provide, in contrast, may be applied to any HSA-compliant architecture. We demonstrate performance overhead reduction from not only register-level communication, but also from the exclusion of memfence instructions, reduced branch divergence, and synchronisation events. To the best of our knowledge, we are the first to implement fingerprinting and effective 32- or 64-bit cross-lane instructions into a fully portable compiler pass for GPU RMT.

Comparative Computing

It is possible to trade off output quality for performance and/or energy in the approximate computation domain. Program modifications for approximation computation that trade output quality for improved speed in an error-free environment. A programming language allows programmers to establish constraints on the chance of error given an output quality operating in unsafe modes. Green suggests an online monitoring system to compromise service quality for energy consumption reduction. In order to achieve speed in place of quality for applications involving image processing, Kulkarni et al. suggest an underdesigned multiplier architecture. A programming language called in supports methodical approximation. It enables programmers to identify individual instructions as essential or non-critical. Using ISA extensions, Truffle, a dual-voltage microarchitecture design, permits mapping of about EnerJ programmes. For important processes, Truffle uses a high voltage (safe mode), whereas for non-critical activities, it uses a low voltage (unsafe mode). Truffles show that adopting dual-voltage operation, which has no cost on switching between safe and unsafe modes for statically partitioned code into critical and non-critical parts, can save up to 43% of energy. By separating control-intensive operations from data-intensive tasks, ERSA separates the execution of iterative algorithms into critical and non-critical code at a finer granularity [82]. Whereas Truffle depends on the programming language support to give safety guarantees, ERSA uses software checks on sub-computations to establish boundaries on execution time and final output and doesn't need recovery for the non-critical computations running in unsafe mode. Relyzer is a resilience analyzer that may assist software developers find sites that are prone to SDCs and trim fault locations up to five orders of magnitude. In order to reveal hardware problems during the execution of unsafe, non-critical code and enable software-based recovery, Relax offers a compiler/architecture system. Relax uses rerun (lowest performance, greatest quality) and trash as a method of programme recovery

(best performance, worst quality). As an extension of a Relax-like system, ASAR is described and offers a user an approximate recovery (good performance, excellent quality) option between repeat and discard. We also provide a hybrid recovery approach that enables considerably finer grained use of the performance-quality trade-off curve.

Architecture of heterogeneous memory

We perceive the possibility for performance and reliability optimization presented by heterogeneity in system design. According to Naithani et al. research, heterogeneous multicore reliability is increased by 60.2% when applications are planned utilising reliability-aware scheduling. In this dissertation, we demonstrate that the 42x FIT rate difference between two memories with different level correction algorithms makes the potential advantages from reliability-aware data locations for memory systems significantly larger. With DDRx memory, reduces the refresh rate for low-risk data. Programmer suggestions must be used to mark low-risk data for Flikker. The demand for additional capacity than what can be provided by the die-stacked DRAM technology now in use is what spurred the development of heterogeneous memory architectures. Typically, the die-stacked component of HMAs is handled as either a large, high-bandwidth Last Level Cache (LLC) or as Part-of-Memory (PoM), in which case the software may access the stacked capacity. We provide performance- and reliability-focused memory management strategies that have been suggested in the literature in this section. Cache organisations for HMAs are investigated in many investigations. Compared to conventional SRAM caches, layered DRAM presents unique difficulties when structured as a cache. Engineers must review and reevaluate cache improvements with DRAM technology in mind, since shows that conventional optimizations designed for SRAM memory are inapplicable for DRAM caches and lead to performance degradation. The creation of intelligent tag stores that prevent repeated cache accesses for tag and data is a crucial DRAM cache improvement.

Die-stacked DRAM caches are proven to only slightly increase capacity-limited applications' performance while dramatically enhancing latency-limited apps' performance. Capacity-limited applications gain in performance when the stacking capacity is instead made available to the application in a PoM arrangement. Mechanisms to handle stacked memory in a PoM architecture have been proposed in recent work. These ideas tackle the memory management issue in a similar way: Data that is anticipated to be hot (often accessed) in the near future should be identified and moved to fast memory. As stacked memory is PoM, some data must be transferred from fast memory to slow memory before being replaced by the anticipated hot data. a migration method that controls HMAs using both SW and HW. A group of access counters that keep track of the access count per page are used to forecast future hot data (4KB pages). The pages with the highest access counts that above a certain threshold are migrated (swapped) into fast memory at predetermined intervals. The OS will carry out the necessary memory operations during migrations in HMA and ultimately update page table entries and TLBs. Migration intervals are maintained long because OS intervention is expensive. A big remap table structure is not necessary to keep track of transferred pages when the OS is involved. It is a newly put forward HW-based memory management method for PoM systems. The suggested design groups fast and slow memory channels into autonomous memory "Pods" and only allows migrations inside those pods. While technically speaking flexibility is still limited, MemPod's design provides sufficient possibilities for pages to move, improving quick memory use. The "Majority Element Algorithm" (MEA) technique is used by MemPod to forecast future hot data. Increased performance is shown to result from using the MEA strategy, which costs a quarter as much as using the HMA's "Full Counters" approach. a hardware memory management system that doesn't have significant space requirements for bookkeeping structures. Memory is divided into "segments," each of which

has a single fast memory address. Memory pages can only replace an existing page by migrating to the segment's fast location. The versatility of this migration process is severely constrained by this segmented approach, sometimes resulting in counterproductive operation: Although the fast memory of a cold segment does not help performance, the fast memory of many hot pages in the same segment will keep pushing each other out of it. Yet, by limiting migrations in this way, it is possible to avoid the difficulty of locating a quick memory location based on potentially more complicated algorithms. tries to manage stacked memory as a combination of a PoM and a cache. CAMEO splits memory into segments similarly to the aforementioned techniques, except it does so at a cache line granularity (64B). Every time a piece of sluggish memory is accessed, data migration begins. Although all of CAMEO's accounting structures are kept in main memory, a lightweight "line position predictor" works to reduce the number of times a line has to be accessed.

The word "IT" covers a wide range of computer-based systems-related disciplines, including software engineering, information systems, computer science, and more. Each of these fields has a distinct concentration and focus. The hardware platform is often seen as an already-existing infrastructure and is sometimes given little attention or understanding. Additionally, recent advancements in hardware technology have mostly focused on identifying the hardware as a distinct platform. As a result, IT staff members, particularly software designers and developers, may see it as a layer of a computer solution that is necessary but which they do not need to comprehend.

It resembles operating a vehicle in that the driver controls it but is unconcerned with its interior workings. On the other hand, recent architectural innovations like cloud computing, virtualization, and the abstractions needed to build contemporary computing systems highlight the need of having a solid grasp of hardware. For instance, desktop virtualization enables access to any programme from any device. The programme will function effectively whether the user uses a desktop computer, laptop, tablet, smartphone, or even a device that hasn't yet been created. Because of this, the focus of this book is not computer organisation but rather current problems with computer hardware and the industry's responses to these problems. From top-down from the application (or programme), which is often created using a high-level programming language like C#, C++, Java, and so on, the majority of the layers in a computer system. In other circumstances, the compiler converts the instructions from high-level programming languages into assembly language, which acts as a mnemonic for the instructions that the computer can comprehend. In other instances, the machine language is directly compiled from the high-level programming languages. The operating system, another software element that is often regarded as a part of the infrastructure, will be able to execute the translated programme (executable) utilising its services. Virtualization, which offers the ability to create virtual machines, is the next level, which is a combined software-hardware layer (this will be covered in more detail in Chapter 10, "Additional Architectures"). The machine instructions are the subsequent level. These binary values serve as the executable instructions and are the only ones that the computer understands. The binary instructions in the majority of contemporary computers are created using predetermined building pieces, often known as 4 microinstructions. The next layer of defines these construction pieces. The digital circuits that comprehend and carry out these building blocks' instructions make up the last level. The dotted line in how the divide between hardware and software used to be rather obvious (Figure 4.2).

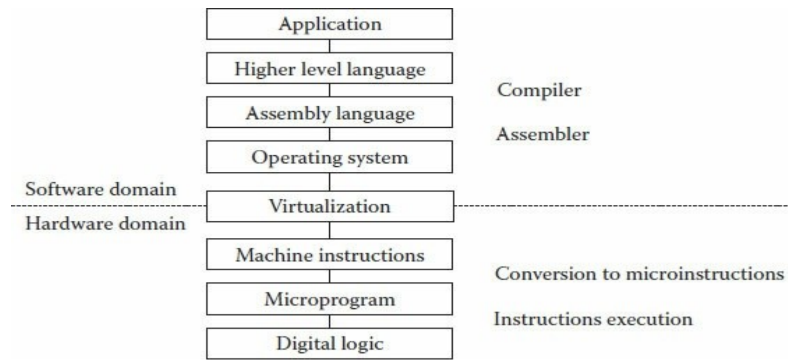

Figure 4.2: Illustrates the System layers.

While the distinction between hardware and software was well established in the early days of computers, developers still needed to be familiar with basic hardware concepts to create dependable and quick-responding software. The hardware and software layers of the solution are still not clearly separated, despite recent technical advancements in cloud computing that further define this boundary. Additionally, software engineering techniques are being incorporated into the hardware development process as more and more hardware components are built utilizing programmable chips. For instance, systems engineering, where a system is often built using subsystems, inspired the modular approach to specifying new instructions using building blocks. Each subsystem is subsequently broken down into ever-tinier components, all the way down to a single function or procedure. As a result, even the second step of Figure 1, which involves establishing the common hardware blocks, is essentially software driven, making the previously distinct division rather hazy. The goal of cloud computing, which has grown in popularity over the last ten years, is to provide a computing infrastructure while lowering costs and enabling users to focus on their core businesses rather than computer-related problems. Virtualization (described in Chapter 10, "Additional Architectures") is one of the key breakthroughs that gave rise to cloud computing. Scalable "virtual machines" are produced by the software layer known as virtualization. The underlying physical hardware is concealed by these devices. By providing a layer of interface, the operating system often bridges the gap between the hardware and software. In order to facilitate access to the hardware, it offers tools and application programming interfaces (APIs) for software development. On the other hand, it makes use of specifically created functions to access the hardware, ensuring a greater level of flexibility and effectiveness. Because to the complexity of software, there is often a need for a layer above the operating system that adds extra common functionality, like Microsoft's.net or Oracle Corporation's J2EE. These environments simplify the software development process by absolving the developer of system duties including memory management, data encryption, and other similar tasks. Instead of "spinning the wheel" and creating libraries of standard services, the developer may focus only on the logic of the application by using the frameworks provided by these virtual environments.

Even while all these new innovations lessen the significance of hardware in software development, it is still crucial to have a fundamental grasp of how hardware parts affect programme execution. A basic grasp of hardware is not necessary to create software; nonetheless, this information is crucial for creating excellent software that is stable and effective. This hardware knowledge becomes essential throughout the first phases of the software engineering process, which determine the functionality of the system. The system analysis step establishes the functionality of the new system once the requirements are established. For a simple software, the operating system, cloud computing, and other development frameworks (like.net and J2EE) may offer adequate degrees of abstraction. Yet,

considerable hardware knowledge is required for bigger software systems, or even organizational software systems, for reasons like feasibility studies and return on investment (ROI) estimates. The "how" is specified after the "what" (what the system will perform) has been established during the design phase (how the system will be implemented). A grasp of hardware is much more crucial at this point. Understanding the hardware components and how they interact with the intended software helps to generate a better design since many design mistakes might have serious consequences. For instance, the majority of current computers use many cores (this will be elaborated on in the subsequent chapters). This indicates that a machine like this may carry out several tasks concurrently. It resembles a dual-line telephone system in that it can accommodate two simultaneous calls. The telephone has the capability, but only two people can use it. While they are able to switch between lines, they are unable to converse with two persons at once (unless it is a conference call). On the other hand, when employing a multi-core system, the programme may be designed using multithreading*, which will more effectively use the hardware platform and improve the user experience while using the newly developed system. An example of a system engineering problem that is both straightforward and basic may be a web application. The outdated, conventional programme supported a single user, received input from the user, carried out the necessary task, created the desired outcome, and then requested the subsequent input, and so on. A web-based application should be built differently if it has to handle a range of concurrent users. A quick job will be created to gather user input requests. This job only stores the requests rather than processing them. The job of retrieving and carrying out each of the submitted requests will be carried out by one or more (perhaps many) additional tasks. This software architecture can be implemented on a single-core CPU, but several cores will make it much more efficient, and in certain circumstances, virtualization will be needed (in which several physical systems will act as a single virtual machine). Large dimensional arrays are another illustration of the significance of hardware knowledge. The order in which an array is indexed may significantly affect how quickly a computer executes computations on an array with two (or more) dimensions, particularly when extremely large arrays are involved. Even on a cloud platform, a program's execution might be significantly sped up by comprehending how memory works and being built properly. Nevertheless, it should be remembered that certain compilers have the ability to alter the code in order to make it take advantage of the particular hardware it is operating on.

Several hardware books provide a very thorough description of the many parts of hardware since they were written with hardware engineers in mind. Nonetheless, this book was intended specifically for IT professionals, for whom different facets of hardware engineering are less significant. The book's objective is to provide a concise historical account of the developments in computer technologies that contributed to the infrastructures that are now in use. The historical viewpoint is crucial since certain phases recur when new electronic gadgets are introduced. The hardware characteristics of mobile devices, for instance, followed some of the advancements in computer systems when the mobile phone revolution first began, for example, regarding memory and its utilisations. However, a lot of the less complex appliances incorporate hardware elements and technological advancements that have been around since different times in the history of computing. For instance, an embedded computer is used in a basic alarm system, but because to its constrained capabilities, it often emulates the design of computers from 20 or 30 years ago. Nonetheless, the parts mirror those seen in contemporary systems (processor, memory, and input and output devices).

Due to the new trend of being connected "always and everywhere," a historical perspective and an understanding of the many phases of growth are crucial. Understanding hardware and

its historical growth is essential, and the Internet of Things, which aspires to link billions of embedded computer devices and build a worldwide interconnected network, adds another dimension to this.

Humans as Computers

An astonishing similarity between the functions of the human body and computer components and architectures is found. The human body, like the bodies of all other living things, is made up of several parts, each of which has a specific function. This modular method is common in the designed products and machinery that surround us, yet it is currently highly restricted for people since only a tiny portion of organs can be replaced. Of course, modern computers provide the same versatility, but the earliest computers were different.

We'll use human processing techniques to better grasp the different computer hardware parts. There are various organs involved when evaluating human thought. The way the brain analyses information is comparable to how a computer's processor works. All actions, as well as behaviour and emotions, are controlled by the brain. This method is built on prior knowledge, whether it was obtained or learnt. Similar to this, the processor responds to the data it receives depending on the loaded software. In this manner, just as experience gives the human brain a guide for behaviour, the programme that is put into the system offers the rules for processing. In this respect, the software engineers' creation may be seen of as training or instructing the computer on how to respond to certain occurrences. This is comparable to instructing pupils on how to behave in a certain circumstance, such as when designing a computerised solution.

The relationship between the human and computer memories makes them even more intriguing. A hierarchy of three layers of memory is how many researchers see the human memory. a temporary memory (also known as sensory memory), which serves as a buffer for data gathered by the body's many senses. Each sense has a separate buffer that stores all of the information it has gathered. Initial processing of this data involves classifying the majority of it as useless and discarding it. The working memory receives the information that may be relevant and copies it for further processing. The enormous quantity of data to which we are exposed as humans serves as one example. For instance, while operating a vehicle, we are continuously aware of our surroundings. These noises and visuals are then sent to the visual sensory memory, where the majority of it is discarded as useless information. In reality, we really don't recall a lot of the sights or sounds we've experienced because to this effort to lessen informational overload. Similar to other living things, people are able to interact with their surroundings and continuously respond to critical events thanks to the sensory memory. These circumstances might provide risks or possibilities to most species (e.g., acquiring or hunting for food).

The memory employed for processing data is known as short-term memory (or working memory). It only gets data that has been deemed significant or pertinent. Here, it is examined to ascertain the right course of action. The short-term memory is reasonably quick to access (tens of milliseconds), yet it only has a little amount of storage space. The amount of time that the information in the short-term memory is stored varies on the amount of new, "essential" information that is being received, the circumstance, and the person's age. Owing to its capacity limitations, this memory is susceptible to environmental disruptions, which may cause the data to vanish, necessitating a particular search procedure (to remember the last thing we were thinking or talking about).

Long-term memory, which is used to store data for an extended period of time possibly

forever. Large quantities of data items may be stored in this memory because to its enormous capacity. The data takes a while to obtain, and as it becomes older, it takes even longer. The techniques for retrieval are unreliable. Sometimes we may strive to recall information but fail. This isn't because a particular piece of data was deleted; rather, a poor retrieval method is at blame. If the missing data is remembered later, it was likely kept in memory and was not lost; it was only that the pointer connecting the data was unclear. Information must be very significant or need continual repetition in order to be kept in the long-term memory.

In relation to computers, all three varieties of human memory exist. Sadly, as will be expounded on in this book, while current computers have a similar memory architecture, it took time before it was created and developed. Computers' long-term memory refers to a variety of storage media, including discs (hard drives). These gadgets are used for long-term information storage. Magnetic tapes, which were formerly common but have mostly been superseded by newer, more dependable technologies and discs, are among the storage devices (magnetic, optical, and electronic, such as solid-state disks). The short-term memory is the memory (or random-access memory [RAM]) and it is the working region. For the CPU to execute a program, it needs to be put into memory as well as the data it requires. While it may contain billions of bytes (or characters) in a contemporary system, this memory often has a restricted capacity. It is still much less than the storage, which may be many orders of magnitude bigger. Computers use registers and cache memory to implement the third level of the sensory memory. The processor has a tiny working area called a register for the data that is presently being utilised and processed. The recently used instructions and data are kept in cache memory, which is a quick but constrained memory. Additionally, cache memory is implemented employing a number of tiers owing to its impact on the system's speed. The cache is quicker and smaller the closer it is to the CPU. The many senses that serve as our input devices (sensors in embedded and real-time systems) and detect the environment, as well as the muscles that regulate movement and speech, which serve as output devices, are the last and most insignificant parts (actuators in real time systems).

The majority of people will read the image as a white triangle perched above three circles. This is an example of an optical illusion since there is no white triangle present; instead, there are just three circles, each of which is incomplete (like the Pac-Man game's eponymous "monster"). The brain's effort to get around a recognised constraint is what leads to the incorrect interpretation. While we live in a three-dimensional world, printed drawings, are only able to provide two dimensions and are thus restricted in their ability to represent the three-dimensional experience. Knowing this, the brain attempts to solve the issue by seeing the as a three-dimensional entity. Without active conscious engagement, this "healing" occurs naturally. The instinctive reactions of all species, such as pulling our hands away from hot surfaces or attacking or fleeing when a danger is sensed, are another illustration of a process that occurs automatically. Without the brain's assistance, these reflexes are controlled, mostly to speed up response to a potentially harmful circumstance. Similar "behaviour" has been acquired by computers using a different method. One such instance is when error correction codes (ECC) are employed in communications to evaluate the accuracy of the messages delivered and received. Before it reaches the user application, it is an automated procedure in which the system makes an effort to catch errors and deliver the appropriate interpretation (just like the Kanizsa illusions). In mistake situations, the controller sends a request again without even consulting the processor. This might be thought of roughly as the controller's reaction to the false message that was sent. The purpose of this behaviour is to eliminate the requirement for the software engineering process to verify the accuracy of the data it receives from other components or during communication.

Computers and other computing tools, which were first made just for counting, have been

needed for as long as there have been people. In the past, people spoke verbally, which led to the development of the necessity for a counting system. A method like this was required to convey concepts relating to quantity, such as the number of sheep in the herd, the amount of time that had elapsed since an event, or the distance to a certain location. The counting process didn't become a system that began utilising symbols to indicate amounts until much later. These symbols sparked a crucial advancement that enabled basic arithmetic operations.

Latin offers an intriguing window into how amounts were represented in antiquity. As the term "digit," the foundation of our numbering system, derives from the Latin word digitus, which meaning "finger," it may be argued that humans originally used their fingers to symbolise amounts. According to this, it stands to reason that the earliest computations were done using pebbles as calculus is Latin for "pebble." The abacus, which dates back to roughly 5000 years and was developed in the ancient civilisation of Mesopotamia, is one of the first calculators ever made. Other civilizations, like the Egyptian, Persian, Chinese, Greek, and so forth, eventually adopted it. In certain far-flung parts of the world, several abacus versions are still in use. The tool was designed to depict amounts as well as aid with basic arithmetic operations like additions and subtraction. Yet, it is challenging to use it for more demanding computations. The next centuries saw the development of a wide range of different calculators. The Napier Bones is one of these tools; it's a small, clever tool used for multiplication and division operations. It was created in the early seventeenth century by Scottish mathematician John Napier, who also made the discovery of logarithms.

The only rods that applied to a certain multiplication were utilised. The actual multiplication is similar to the usual multiplication method taught in primary schools today, which begins with the smallest digit and moves up to all higher-order numbers. Assume for the moment that we wish to multiply 3789 by 8 using the Napier Bones. The rods that must be chosen and utilised are 3, 7, 8, and 9. When multiplying from right to left, we begin by putting down the values in row 8. The units digit will be 2, and the tens digit will be 1 since 4 + 7 = 11, which is bigger than 9, causes a carryover and we only record the units digit. By combining the two diagonal digits and the carryover, 6 + 6 + 1 = 13, which results in an extra carryover, the third digit (hundreds) will be 3. The final digit (ten thousands) will be 3, and the fourth digit (thousands) will be 0 (5 + 4 + 1 = 10). The answer is 30312 (3789 multiplied by 8 equals 30312).

The discovery of logarithms considerably reduced the complexity of computations by substituting additions and subtraction for multiplications and divisions. The behaviour of logarithms served as the foundation for the slide rule, which was developed in the seventeenth century. The slide rule, which has been referred to as a mechanical analogue computer, was used by scientists and engineers for more than 300 years until being progressively displaced by the electronic calculator. The rule was based on two sliding logarithmic scales that could be lined together to determine the outcome of the multiplication or division in its most basic form.

There are various ways to simplify expressions using logarithms. For example

$$\log(xy) = \log(x) + \log(y)$$

and $\log\left(\frac{x}{y}\right) = \log(x) - \log(y)$

By shifting the top scale to the right up to position x on the lower scale, division actually aligns the two slides, one after the other. The number on the lower scale that scales with the y on the upper scale is the end outcome. Let's say, for instance, that we need to multiply 2 by 4.

Then, we align the top scale such that 1 and 2 on the lower scale line up, and then we align scale 4 so that it lines up with the results. Blaise Pascal, a French mathematician, added to the race to create a tool that would streamline mathematical operations. This machine functioned as a mechanical calculator and used a system of linked wheels, which is a design seen in mechanical clocks.

Charles Babbage is regarded by many to be the true "father of the computer," however. He was a mathematician from England who created what is thought to be the first mechanical computer. Even if it didn't work out, it opened the door for other, fresher concepts. Babbage, like his forebears, was interested in finding ways to replace difficult mathematical computations like polynomial functions with a set of straightforward operations utilising only additions and subtractions. This is because additions and subtractions can be relatively easily implemented using a mechanical wheel. The gadget was known as the difference engine because it eliminated the need for division and multiplication via the use of finite differences.

Babbage never really constructed the differences engine, although having planned it. Henry carried on his father's research and created a gadget out of components he discovered in the lab. Nevertheless, it took an additional 150 years to finish the gadget because better manufacturing techniques and construction materials were used. Many machines were created, all of which were based on Babbage's initial sketches.

CHAPTER 5

BUILDING BLOCKS OF CONTEMPORARY COMPUTERS

Jagdish Chandra Patni, Professor,
Department of Computer Science and Engineering,
Jain (Deemed to be University) Bangalore, Karnataka, India
Email Id- jagdish.cp@jainuniversity.ac.in

Babbage himself came to the conclusion that the differences engine cannot be used to produce the "actual" solution for a computing device, thus he abandoned this notion and began developing an analytical engine. The basic building blocks of contemporary computers are a processing unit, a control unit, memory, and input and output devices, even though the analytical engine concept was abandoned before it ever got off the ground. The 1000-cell memory of Babbage's analytical engine could store up to 50 numbers in each cell. The analytical engine included a method for writing out part of the cells to make room for fresh numbers, which allowed for greater memory use. Most general-purpose computers that employ virtual memory and other storage devices heavily use this concept. The knowledge that the machine should be able to run other programmes in addition to the one it was intended for was the analytical engine's most crucial component, however. One of the special qualities of computers is this knowledge, which subsequently led to the creation of programming languages. Computers and other devices based on computers may be made to perform a variety of different tasks by isolating the software or run programmes from the hardware. Computers may be configured to behave differently by loading various apps, unlike all other devices or appliances that are built to carry out a certain duty. The computer (desktop, laptop, tablet, etc.) waits for the user's commands after starting. The computer "transforms" into a word-processing machine when a word-processing programme is opened. When the user launches a browser, the computer "transforms" into a tool that enables web browsing. But, if the user chooses to start a game, the computer transforms into a gaming console. Nevertheless, it should be remembered that only the programme that is loaded and run changes, not the computer hardware. The ability of computers to take on many different identities is made possible by the division of hardware and software, which was first conceived of by Babbage in his analytical engine.

Computers are widely used in other devices and have several functions, therefore multifunctionality and identity shifts are not limited to computers alone. For instance, modern mobile phones have many CPUs and provide much more than their primary and original function of making phone calls. These diminutive devices have evolved into mobile personal information hubs that offer web connectivity, a local database for various types of information (contact lists, photos, texts, videos, music, etc.), and an entertainment hub for playing games, movies, and other media. By adding additional applications, the phone's identity is further altered.

Initial computers

Many people refer to Herman Hollerith as the "father" of modern computers since he developed a tabulation device based on punched cards in the late nineteenth century. This is mostly because of his useful contribution to the advancement of the topic. Technical advancements were motivated by the desire to address a specific issue, as has often happened

in the past and in the present. Several waves of immigrants arrived in the United States in the second part of the nineteenth century in search of better lives. Given that the U.S. constitution mandates a census every 10 years, these hundreds of thousands of additional residents necessitated one. At the time, technology was relied on human registration. Such a census would have taken a very long time to complete with the current technology because of the size of the United States and the quantity of persons involved. The census was used to capture significant information conveyed via questions in addition to population counts, therefore extra tabulations were needed for the solution. The hand-tabulated 1880 census took over ten years to complete. The primary issue with manual technology was that, by the time it was finished, the findings were inaccurate and sometimes even irrelevant. This was in addition to the very lengthy time required. It was obvious that a different method would need to be employed for the 1890 census. Of course, it needed to be speedier and more dependable. For a remedy to the issue, the census office published a Request for Proposals (RFP). In the present day, we speak about a new technology, although the census office was searching for a novel census method in the nineteenth century. Herman Hollerith's winning idea was based on a device that could read data from punched cards. By poking holes on the cards, the information from the census was recorded. The devices used a metal brush to "read" the data. By passing through the holes in the card and completing an electrical connection, the brush detected the holes in the card.

The statistician Hollerith didn't want to create a new technology; instead, he wanted to create a tool that could manage enormous volumes of data, particularly when it came to organising and tabulating such data. Computers continued to utilise the newly developed punched card technology for many years after that. Punched cards were one of the original input/output devices, and they were extensively used for storing and inputting data into computers throughout the first part of the 20th century. Punched cards weren't superseded by other, more advanced technology until the 1980s.

The 1890 census took roughly six months to complete and cost $5 million (U.S.) less than expected. The first instance of a computer replacing people in handling enormous quantities of data, which documented Hollerith as one of the founders of modern computing, is the most significant problem, not the technology or the time and money it saved. Because of the technology's widespread usage, Hollerith established the Tabulating Machines Corporation, which won the 1900 RFP for the census. By combining two additional businesses in 1911, it expanded and became CTR (Computing Tabulating Recording Corporation), which underwent a name change to IBM in 1924. (International Business Machines). There were 12 rows and 80 columns on the Hollerith punched cards that were in use up until the 1980s. Up to 80 characters (numbers, letters, and special symbols), one per column, may be stored on each card. Numbers 0 to 9 were written on ten rows, while rows 11 and 12 were left empty. Whereas other characters were encoded by a mixture of holes, digits were represented by a hole in rows 0–9 that denotes the digit (0–9 and 11–12).

While primarily employed for information storage and tabulation, Hollerith punched cards enhanced technical state-of-the-art and subsequently became the principal source of input and output media. The Mark I, an electromechanical computer created by Howard H. Aiken and sponsored and constructed by IBM, was delivered to Harvard University in 1944 and marked the beginning of the modern age of computing. As it employed mechanical relays, the computer was very loud and had a length of almost 50 feet. The architecture was based on Babbage's analytical engine, however an electrical motor was used to carry out the notion. The Mark I employed decimal numbers instead of binary arithmetic, and its execution speeds were poor even when compared to those of the human brain:

1. Every second, three additions and subtractions
2. For multiplications, four to six seconds
3. for divisions, fifteen seconds

The data was input via manual switches that represented the numbers, and the execution instructions were read one at a time from punched paper tape.

Harvard and IBM both created new and more sophisticated computer models based on the knowledge gained from the Mark i. The Electronic Numerical Integrator and Computer, however, was a significant turning point in the development of computers (ENIAC). The first mechanically free electronic computer was the ENIAC. It is regarded as the very first general-purpose computer because of its capacity to execute a wide range of applications. Its principal objective was to make the laborious process of calculating and creating artillery-firing tables easier. John Mauchly and Presper Eckert oversaw the machine's design and development at the University of Pennsylvania. The project began in 1943, and the computer wasn't fully functional until 1946. While it was too late to help with the war effort, because it remained in service until 1955, it was utilised for many other labor-intensive computing projects, including the hydrogen bomb.

Like all other computers of the time, the ENIAC was enormous, covering 1800 square feet. It used more than 17,000 vacuum tubes, which were not very dependable at the time. As a consequence, the computer remained inactive for a considerable amount of time and was only active for half of the time. While it was functioning, nevertheless, it was the fastest computer on the market, able to do 5000 additions/subtractions, 357 multiplications, and 38 divisions in a single second. The knowledge that a computer should be a general-purpose device capable of running a wide variety of applications is the trait that advances computing technology the most. Nowadays, one of the major tenets of computer designs is this idea. It influenced the creation of several programming languages and significantly increased the use of computers both as standalone systems and as embedded software in various machines, gadgets, and appliances.

Features of the First Computers

The field of computing did not emerge overnight; rather, it evolved gradually. The early computers were distinct from others that came after them and much different from the ones we use today. The early computers were made to function in a similar manner since the decimal system was used by the majority of civilizations. It wasn't until much later that it became clear that a binary system is considerably more effective and suited for electrical equipment like computers. Size is a second important distinction. In the early days of computing, electronics and minimising technologies were not as advanced, which resulted in enormously massive computers that used a lot of power and had a narrow range of solutions they could provide. The transistor's creation marked a tremendous advancement (1947). A semiconductor device that can switch and amplify signals is the transistor. The electromechanical switches and vacuum tubes, which were much bigger, used a lot of power, and weren't very dependable, were replaced by transistors. The development of contemporary technology took a significant stride forward with the advent of the transistor, which permanently altered the electronics industry. The transistor enables the integration of millions, and in more recent years, even billions, of transistors on a single chip, as well as the reduction of electronic circuits. The computer sector as well as the whole electronics market underwent a revolution because to these characteristics.

In the early computers, "programming" was carried out by reading punched paper tapes or using manual switches. After reading and carrying out the first command, the computer

moved on to the next. Only later did the concept of loading a programme into memory or the "stored programmes paradigm" emerge. The inclusion of registers was another significant modification. Registers are quick buffers used by the processor unit to store transient data. Registers are the comparable hardware part to sensory data in the human body (see the preamble). Registers are a crucial component of the execution of the instruction, even if they are not utilised to store all sensory input and ignore the extraneous portions (as in humans). Temporary variables were stored in memory or the stack on the first computers since there were no registers. The one-register (sometimes known as "accumulator") computer was not created until much later. This accumulator was used to store intermediate findings for use in computations of greater complexity. It functioned in a manner similar to the basic calculator's display. Subsequently, the significance of registers was understood, and modern computers now have hundreds of registers to help with execution speed.

Von Neumann Building

John von Neumann, who worked on the development of the hydrogen bomb, was also somewhat engaged in the ENIAC project, mostly because of its potential for handling the challenging calculations required for the weapon. Von Neumann contributed to the creation of a computer architecture that is still used in the construction of contemporary computers. In actuality, system engineering also benefited from the use of the modularity concept later on. The shared memory of the von Neumann architecture is utilised to store both data and instructions and is comparable to human short-term memory. The model identifies a number of various but linked elements that make up the computer architecture. The model is known as the stored programme model because it differs from the earlier models' single programme availability by allowing several programmes to be loaded into the computer's memory. Modularity is a crucial concept that emerged from the von Neumann architecture and enhanced computer utilisation while bringing down their cost. The model specifies many functional components that are clearly separated from one another. For instance, separating the memory from the computing unit. Executing the instructions is the responsibility of the processing unit (also known as the processor), which has nothing to do with where the data or the instructions are stored. The processor is the human body's version of the brain in computers. The instructions and the operands necessary for their execution are fetched by a specific component inside the control unit of the processor. The result of the performed instruction is gathered and stored in the defined place by a comparable component (Figure 5.1).

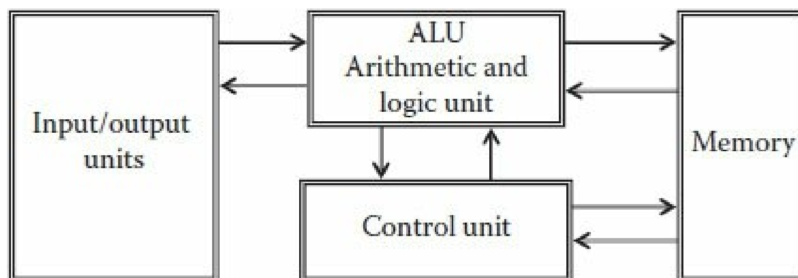

Figure 5.1: illustrates the von Neumann model.

According to the figure 1 describing a special method for both storing and retrieving data from memory. The essential idea is that a computer system may treat programmes as data and load them into memory to run them. The foundation of the stored programme model is this technique. Modern computers were made possible by the separation of the memory from the processor and the realisation that data and programs are variables that may change. These

computers have the capacity to execute several applications simultaneously. These applications alter the system's operation in accordance with their own specifications. Also, the computer's capacity to execute many programmes or apps simultaneously enables it to provide various functionality to various users as well as to the same user in various windows. dividing the control unit from the execution unit. The execution unit, which is in charge of carrying out the instructions from the programmes, was originally formed by fusing the two units. After the separation, the control unit is in charge of planning the execution and supplying all of the operands and instructions required for the execution, while the execution unit just carries out the instructions. segregating each unit from the other units as well as the input and output units from other system components for instance, the great level of modularity we all enjoy with personal computers is a byproduct. Owing to this division, one may purchase a computer from a manufacturer while the mouse and keyboard, as well as any other input and output devices, may be purchased independently and linked to the computer. Other electrical devices seldom exhibit this high level of modularity, unless they include embedded computers. Electronic components in devices may be changed, although generally with other original or compatible components. These new parts must be completely compatible and provide the same functionality. The user may swap out a component, such a hard disc, with an entirely other device using the computer's standard interfaces. Installing a new, speedier solid-state drive or, alternatively, a different rotating hard drive with a much bigger capacity may be done in place of a spinning hard disc.

The Institute of Advanced Technologies (IAS) used the von Neumann architecture for the first time (at Princeton University). In contrast to the earlier manual switch settings, the IAS computer could load programmes. The computer utilised binary integers, had multiple registers, and was specifically created for doing intricate mathematical calculations.

Development of Computers

Even experts were surprised by how quickly technology developed and advanced over the last several decades. Several people who sought to predict future computing trends grossly misjudged computers' impact on our lives and their contribution to our society. Some of the most well-known forecasts are as follows:

A Popular Mechanics article from 1948 claimed that computers of the future will likely weigh no more than 1.5 tonnes. The article referred to this as a remarkable advancement, but it did not anticipate that computers' weight would continue to decrease by three or four orders of magnitude.

One of the cofounders of Digital Equipment Company and computer sceptic Ken Olson argued there was no justification for anybody to desire a computer in their house in 1977.

One of the key factors in the quick advancements was the introduction of the transistor, which had an impact on the whole electronics business in addition to computers. The combination of billions of transistors on a single chip and their massive downsizing cleared the door for a never-ending stream of new generations of computers, each one quicker than the last, with more sophisticated features and less power usage.

But it's important to remember that the increase in transistors isn't only tied to the processor; it's also related to the new features that were introduced to improve its usefulness and performance. Cache memory, which is discussed in more detail later in this book in Chapter 6, is a crucial element in boosting the machine's execution speed. Also, current innovations focus expanding the processor's core count, which results in significant performance benefits. The number of transistors on a chip is increasing, but different strategies are being investigated in response to global developments like the energy crisis and the enormous need

for portable and electricity-efficient CPUs. An enormous amount of work is being spent towards lowering the necessary power consumption and heat dispersion rather than boosting the processing speed (Table 5.1).

Table 5.1: illustrates the Number of Transistors

Chipa	Clock speed		
4004	108 kHz	Year of introduction	Transistors
8008	200 kHz	1971	2,300
8080	2 MHz	1972	3,500
8086	4.77 MHz	1974	6,000
Intel286™	6 MHz	1978	29,000
Intel386™ DX processor	16 MHz	1982	134,000
Intel486™ DX processor	25 MHz	1985	275,000

For instance, Mr. Urs Hoelzle, Google's senior vice president for technical infrastructure, provided some information regarding Google's power use in a story that appeared in the New York Times1 on September 9, 2011. 260 megawatts are continuously used by the Google data centres that power searches, YouTube videos, Gmail, and other services. Also, Google said in another article2 that was released in September 2013 that it will purchase all the power produced by a 240 megawatt wind farm constructed in Texas. These two instances show how the new research paths are justified. Significant research is being done to create new, lighter, quicker chips that use less energy, in addition to the efforts being made to develop new methods of primarily improving processing speed. During their assessment of the condition of the personal computer (PC) market in 2008, Gartner, a renowned information technology research and consulting firm, predicted that there were more than 1 billion PCs in use globally3. By 2014, according to Gartner researchers, the installed base will have surpassed 2 billion globally. More thorough shipping information was made available in a recent Gartner4 report. Its study predicted that conventional PC shipments would decrease over the next years while mobile phone sales would increase. These new trends have an impact on the competition to pack more transistors onto a chip because they are driven by new users' needs for continual network connectivity. Mobility, portability, and low power consumption have grown in importance and must now be taken into account while designing gear.

Moore's Rule

Gordon Moore, who was Fairchild Semiconductor's head of research and development (R&D) at the time, made an intriguing prediction concerning the number of components in an integrated circuit in an article that was published in Electronics on April 19, 1965. Based on extrapolating the quantity of components being utilised at the time, it was predicted that the number of components per circuit would quadruple every year. Moore revised his prediction to a doubling of the number of components every 24 months. Moore would eventually become one of the founders of Intel. Hence, Moore's law5 is an observation rather than a physical law that has influenced the semiconductor industry and all gadgets that employ these circuits. Moore's law is significant because it not only affects the quantity of components used in integrated circuits but also because the efficiency of these devices is directly correlated with the number of components used. Contrary to popular opinion, David House, an Intel executive at the time, modified Moore's original assertion that the number of components would double every 18 months. House talked generally about the performance of the computers rather than precisely about the number of components in the circuit. Moore's

law, which mostly applies to hardware, has an impact on the system engineering and software development processes as well. Due to the fast developments in computing technology foreseen by the legislation, even issues that now appear intractable because of a lack of processing capacity will be resolved in a few years.

Computer classification

Computers may be categorised in a variety of ways, but throughout time, the most common categorization has been based on the systems' intended application. There are three major categories of computers and computer-based systems, roughly speaking:

Microcomputers, which are often portable, inexpensive, and designed for a single user or device. Personal computers, which typically come with a monitor, keyboard, and mouse and are used for a wide range of tasks including network access, business, educational, leisure, and even gaming, are found in almost every workplace and home. PCs are now the most popular platform for application development and solutions across a broad range of industries thanks to their widespread use and recent yearly sales of over 300 million units. These Computers serve a wide range of functions and use a wide range of hardware and software components. While most modern PCs are laptops, a subset of microcomputers has been established for extremely specialised needs.

This subcategory took the form of tablets and PDAs (personal digital assistants), which make use of alternative input methods like touch screens, which are often built-in to the device. Contrary to this, the architecture of typical PCs facilitates the intermixing of input and output devices and is built on modularity.

The present engineering efforts are mostly focused on these new mobile devices and contemporary mobile phones because of their enormous potential (as shown by the global device shipment—Table 1.3). Also, all the capability that was previously exclusive to desktop and laptop computers has been made accessible on mobile phones thanks to the convergence of mobile networks and the Internet. The incorporation of these mobile devices into the operational computer network is necessary due to the fact that a large portion of the workforce in many firms is mobile (travelling sales representatives, service and repair personnel, appraisers, etc.). This integration is crucial for improved managerial control and more informed, fact-based decision-making. The many delivery services like FedEx, UPS, and others who have given all of their dispatchers personal devices are a very apparent example. Every action, including picking up a package, putting it on an aircraft, carrying it to a delivery location, and so on, is captured and sent in real time to the control centre. Each client may trace the whereabouts of his or her package nearly in real time thanks to the website's integration of the information system at the control centre. The market for different embedded devices is another extremely sizable business area that makes use of several microcomputers. The need for incorporating a computer develops when most of the objects and appliances that around us grow more intricate and extremely smart. The majority of household appliances in the twenty-first century use some kind of digital technology. Vehicles, commercial equipment, medical devices, and other items fall within this category as well. These gadgets are all built around microcomputers.

Minicomputers, which are often utilised in environments with several users. Traditionally, rather from being created to serve the needs of whole businesses, these computers were designed to address the needs of a single department. These minicomputers might serve a geographically isolated department before network technology advanced. Moore's law's prediction of fast improvements in PC performance has caused minicomputers to evolve into tiny servers in recent years, sometimes using a conventional PC. These servers serve as the

foundation for a particular application, such as a print server that controls all printers and manages all print-related operations or a mail server that controls all of the organization's communications.

Large corporate computer systems known as mainframes serve as the primary computational backbone. These systems offer applications to support all business activities and serve as the host for an organization's databases. Typically, mainframes are quite powerful and can accommodate several concurrent users. The first mainframes had proprietary hardware and an exclusive operating system. Large servers, which provide the same functionality as mainframes, have mostly taken their place since the turn of the twenty-first century. Several of these servers are constructed utilising many commercially available microprocessors operating in parallel. New server farms were constructed to house several servers operating and sharing the same facility in order to supply the necessary power and functionality. These farms provide a higher degree of parallelism and the possibility of infinite computing power. Also, as computer technology advances, tools for defining virtual machines are become available. Virtual machines are often composed of several physical systems that operate as a single virtual system. Several contemporary developments make advantage of this virtualization in conjunction with quick connections to provide cloud computing. An business may purchase computing services from a third-party provider using cloud computing technology; more on this will be covered in Chapter 10, "Additional Architectures."

Perspective from the past

Midway through the 20th century, the first computers were constructed. They were both physically and financially enormous. Due to the heat created by the vacuum tubes, which were a primary factor in these systems' cooling facility needs, they employed proprietary operating systems. These computers only supported batch processing and could only execute one application (or process) at a time. The computer software operates offline during batch processing without any user input or interaction, in contrast to the contemporary interactive mode of operation. The program's necessary data is given in advance. The operating system was in charge of executing the batch programmes one at a time once they had been put into the system. The fact that the system only ran one programme at a time meant that it was necessary for the current programme to finish executing before the next programme in the queue could begin. This way of doing things was very ineffective. The input and output (I/O) devices were much slower than the CPU even in the early days of computing. The CPU had to wait for the I/O operations since only one application was running at a time. Overall, it was determined that the system was not only exceedingly costly but also inefficient due to this manner of operation. This was particularly apparent when the software needed to handle several inputs and outputs.

These restrictions served as one of the impetuses for attempts to enhance computer technology, primarily by introducing tools for running many applications simultaneously. They were initially batch operations, and interactive sessions were added only afterwards. The processor might be used to run another application while one was waiting for an event, such the users submitting some input. Certain hardware changes were needed to permit the execution of many programmes simultaneously, such as to provide the necessary isolation between the executing processes. In order to safeguard each programme and its working area from illegal access and to ensure that each programme was only accessing its working space, the operating systems had to be updated as well.

Users benefited greatly from working interactively since they could obtain the needed response right away. As a result, batch systems gave way to more interactive ones. The significant disparity between CPU speed and I/O device performance gave rise to the time-

sharing systems now employed by the majority of computer systems. The operating systems were changed to accommodate a large number of concurrent users who share the computer system since the processor was fast enough to handle multiple interactive clients. Each user believes that they are the only ones being serviced by the system, yet in reality, all users may be supported by it. This naturally necessitated further adjustments and improvements to the operating system in terms of scheduling the algorithms in charge of giving each working client a fair part of the system. Despite their very high cost, computer systems were widely adopted due in part to their capacity to serve many concurrent users. Time-sharing computing was originally implemented using "dumb terminals." These were straightforward tools for interacting with the computer system. These terminals had a keyboard for entering data and a printing system for showing the computer's responses. The output device wasn't replaced with a screen, which is what most current systems utilise, until much later. These terminals were entirely mechanical, and the computer received every character typed on the keyboard. The computer instructed the printing device to print the character that was input. Each of these terminals was initially linked to the computer by a separate communication connection. Also, each manufacturer had pricey proprietary terminals that were created specifically for their computer.

Computing departments tried to load the processor to its utmost capacity in an effort to justify the high expenditures connected with computers. Nevertheless, a fully loaded system operates more slowly, which many users found intolerable after first experiencing quick response times. Moore's law's prediction of rapid technological advancement led to the creation of new generations of computers that were physically smaller, often slower, and didn't need specialised cooling systems. It created a fresh market for computer makers. Computers were made available to departments when the cost of a computer system dropped from millions of dollars to hundreds of thousands and then to tens of thousands. Smaller departmental computers began to replace the huge computer centre run by the computing department. As the department's users did not have to compete for computer resources with everyone else in the business, the departmental computer had the principal benefit of improving response times. These departmental computers often ran a customised operating system and made use of unique proprietary terminals.

Departmental computers began to arrive at the same time as communications technology, another important technical advancement, started to take shape. Initially, each terminal had its own dedicated line that was used to connect to the computer. The offices of the users, however, may sometimes be found at a remote location. In many instances, many terminals had to be linked at such a far-off workplace. It was exceedingly costly and unnecessary to use a separate line for each terminal, particularly when there was very little use of these lines. A concentrator, a compact communication device that can transport data generated by several sources operating simultaneously over one communication line, was proposed as the answer. Similar to how the computer's CPU shares time, the concentrator believes that communication lines are quicker than user interaction. As a result, sharing a communication line among many users results in large cost savings with little drawbacks. The company might expand the number of terminals at faraway sites by employing such concentrators without building new communication lines and by using the current infrastructure. Adding a terminal at a distant location is often less expensive than installing one locally.

An isolated location with four terminals is shown in the top portion. Prior to the use of concentrators, each terminal required its own communication connection to be deployed. The concentrator reduces expenses by 75% by using one line for all four terminals. There are extra ongoing maintenance expenditures on top of the one-time costs related to such a deployment. Often, an organisation cannot install communication lines off-campus, therefore

an outside source of communications is required. For its services, such a company often levies a maintenance fee. These communication expenses are also decreased by a concentrator that allows many logical data streams to be supported on a single physical line. On each line, there are two concentrators that manage the data stream invisibly. Each terminal on the other side and the computer on the opposite side are unaware that they do not share a dedicated line.

Individual computers

Microcomputers first appeared in the early 1970s and are distinguished by a single-chip CPU. The Intel 4004 was one of the earliest microcomputers, debuting in 1971 and used 4 bits. Many businesses investigated the potential of microprocessors for diverse uses in the 1970s. The software systems created in the early part of that decade were quite specialised, such as an engineering gadget. Nevertheless, additional "off-the-shelf" applications, such spreadsheets and games, began to emerge in the second part of that decade. To save expenses, certain microcomputers from the 1970s, particularly those used for gaming, were coupled with a television.

Early in the 1980s, IBM quietly assessed the possibilities of the "new" microcomputers. There were also reports that IBM planned to acquire the highly skilled arcade business Atari. With a microcomputer-based device devoted to visual games, Atari made its consumer market debut in the early 1970s. The Atari 800, Atari's most well-known game console, had a rudimentary programming language and served as a straightforward and reasonably priced personal computer. The arcade game industry accounted for a significant amount of the continuous expansion of the microcomputer market. IBM made the decision to join the industry by creating its own hardware platform after seeing the enormous market potential. The first IBM microcomputer was released in 1981. It was built on an 8088 CPU, included a floppy disc for data storage, and had 16 KB (kilobytes) of memory that could be expanded to 256 KB. It was given the moniker Personal Computer, which is still in use today. However, IBM's entry into the market gave the technology the necessary respectability, and over time it also started to be used as a platform for commercial applications.

Being the first complete computer to be built entirely from commercially available components without any proprietary IBM inventions, the personal computer was a crucial turning point in IBM's history. Additionally, as the computers were offered in standard commercial retail chains, even the marketing campaigns for the new computer were creative. A tiny firm led by Bill Gates created the operating system for the new machines, and he was successful in convincing IBM to let him to market the software to other businesses. The personal computer was named the Machine of the Year by Time magazine a few months after its debut. This served as a monument to the PC's accomplishments, particularly its potential for improving human welfare. In reality, the PC sparked a revolution in computing that continues to influence contemporary systems. These computer designs often use mobile, lightweight versions of the Personal Computer in addition to it.

The first PCs were quite basic and incredibly unsophisticated. The PC was initially designed for a single user and could only execute one programme at a time, much as the first huge computers. Yet, PCs also incorporated technology advancements for powerful computers, which steadily improved their capabilities. The early PCs were used for stand-alone issue resolution, such as spreadsheets that managers utilised as part of the management process or simple apps for task management and tracking. Nonetheless, the PC could run other programmes as well, just like any other computer. Emulation of proprietary terminals was one of the first applications created and was quite successful. At that time, only proprietary terminals created and produced by the hardware vendor were utilised with organisational

computers. This implied that the seller alone set the pricing for these terminals. Customers were forced to pay the needed price, even if it was exorbitant and unfair, since there was no competition. The compatible devices were offered by the PCs that could run a programme that replicated these proprietary terminals, but at a much lower cost. The costs of the proprietary terminals fell dramatically in a short period of time. Nevertheless, when other firms joined the market and produced their own version of the computers, the number of PCs made and sold increased, which also resulted in a fall in PC pricing. In hindsight, the primary reason there are today so few proprietary terminals is the tendency of utilising a PC with an emulation programme to replace the proprietary terminals.

Systems with some proprietary terminals and some PCs were the direct outcome of bringing PC-based terminals into enterprises. Such a PC may also be used for various purposes other from serving as a terminal. Several of the PCs were operated independently from the primary computer and were utilised as standalone systems. The PC gradually but firmly cemented its place in businesses. Its popularity was boosted by the affordable pricing and the wide selection of apps and games offered. On the other side, as more systems were sold, their market value dropped. PCs began to emerge in every business and the majority of homes. The cost for each Computer at one point reached a stable level of $1000. For many years, this price stayed constant, and as Moore's law predicted, waiting the purchase gave consumers a quicker system for the same cost. The price of PCs didn't start to fall further until the first and second decades of the twenty-first century, and this was due to a new generation of personal electronics like smart phones, tablets, and other gadgets. Without investing in research and development, the technology used in PCs was basically a copy of the technology created for huge computers. This fact, together with the huge number of PCs being produced annually (hundreds of millions), was the primary driver of the sudden drop in PC pricing. The PC was able to effectively compete with huge systems that were many years older and several orders of magnitude more costly for several years after its release. It should be emphasised that the systems' speed is just one factor used to evaluate their performance. Computers' capacity to sustain massive volumes of information transfers was compromised in order to decrease the price and because they were originally designed for a single user. Hence, even while a PC is capable of running a particular application just as a huge system is, it falls short of these giant computers or contemporary servers when it comes to managing high volumes of traffic or demanding input and output needs.

CHAPTER 6

NETWORKS OF COMPUTERS

Mahesh T R, Associate Professor,
Department of Computer Science and Engineering,
Jain (Deemed to be University) Bangalore, Karnataka, India
Email Id- t.mahesh@jainuniversity.ac.in

The PC's capabilities grew as time went on as well as its development. It got quicker, could accommodate bigger memory, and offered more capabilities. The PC gradually replaced other forms of infrastructure as the primary platform for small business applications, enabling many concurrent users. Larger files have to be supported by Computers as a consequence. Computer networks began to develop to better meet this demand. These networks were created so that several PCs may join and share hardware resources while operating in a shared workgroup. The following were the factors that supported the growth of networks: The need to access shared files Every organisation keeps shared files that store or manage organisational constants, like VAT or holiday dates, etc. Typically, the accountable party will change the value in a particular table or file, which all users and programmes may access. Replications of the file will be required if there is no network that enables all computing devices to view the particular file. One duplicate will eventually be forgotten in such a situation, leading to inconsistent and untrustworthy outcomes. A system that enables all computing devices to access a single file is the best and simplest solution to avoid this scenario.

Exchanging files the desire to share the work completed by numerous users developed after the PC became a widely used working platform. In today's world, many projects are created by teams, and under these circumstances, it is necessary to share ideas and incomplete work. Initially, the group members shared a common file in order to do this. A network must be established, even with this option. At the second stage, teamwork was required, as well as cooperation between members of several teams that were working on the same project. A data communication network is the only workable approach due of the teams' sometimes separated locations.

Primary backup: The early Computers sometimes had a variety of dependability issues. The users lacked sound backup processes because they often lacked the necessary computer competence. Following several incidents when users lost their data, which often resulted in numerous wasted hours of labour, it became clear that a dependable backup (and restore) system should be put in place. The best answer could be offered by an organization's computer department since it possessed the necessary skills, as well as the right equipment, methods, and hardware. While it is feasible to set up backup processes for every PC, giving the task to the computer department is safer, more cost-effective, and more effective. Also, the different backup devices were not widely accessible for PCs since they were often attached to huge computers. Information—and sometimes even crucial information—was saved on Computers as PC use expanded in many enterprises. No company will tolerate the possibility that important and historic data might be lost because users lack computer expertise. This factor alone is a big facilitator for networks implementation.

Sharing costly and distinctive peripherals: Before businesses predominantly used central computers, it was conceivable to buy an expensive or unique peripheral that could be used by all users. For instance, many individual users can now buy a laser printer, but in the early

1980s, such equipment were quite costly. The majority of peripheral devices may now be linked to the Computer thanks to the essential advancements brought about by the PC revolution. Nevertheless, a network is necessary in order for other Computers to access the gadget. Additionally, it makes little sense to include even common, inexpensive I/O devices like scanners and printers, much alone more costly ones, in every Computer. The majority of businesses provide these peripheral devices per set of Computers, and a network is needed to access them.

Employing a network: The need of working in a linked environment was the problem that largely influenced the creation of networks. Without an underlying network, it would have been impossible to construct e-mail systems, chat features, message boards, forums, and other communication tools. It is now impossible to envision contemporary life without internet communication since it has become such a fundamental component of it. Only once networks were created did all of these powers become a reality. They were first created to link PCs within of companies, and subsequently they were expanded to link PCs outside of those organisations and at distant regions.

Remote work: Even though a distant computer is physically far away, one person may access it over the network and do tasks on it. While working on a distant system wasn't conceivable in the early phases of network development, this functionality was created to address a genuine need. Remote work is different from collaborative work (previously mentioned) in that it involves working on specialised computers or systems with specialised software installed, such as parallel systems, which have several processors and are used to execute very big programmes quickly. For instance, a very big model has to be examined in order to provide an accurate weather forecast. A sizable and often enormous parallel system is employed to provide the forecast in time. The European Center for Medium-Range Weather Predictions (ECMWF), which is based in the United Kingdom and is partly sponsored by the European member states, is one example of a weather centre that uses a common system since not all of them can afford such a system.

The Defense Advanced Research Projects Agency initially developed the Internet as a military research and development project before it gained widespread recognition in the late 1980s (DARPA). Many hardware and software tools were created as a result of the capabilities of the new network and its potential applications. These technologies made it possible for PCs to participate in the network and take an active part in many of the services being offered. Nowadays, a lot of programmes use the Internet, unless doing so is forbidden for security concerns.

It is important to remember that the early phases of networking mostly included connecting one computer to another or a group of computers to a server. Wide-area networks (WAN), which link systems globally and enable Internet access, were invented much later. In addition, many other kinds of gadgets, such web cameras, vending machines, elevators, GPS devices, and so on, utilise the Internet, which serves as a worldwide infrastructure for connection.

Computers in the 1970s: The First Mainframes

In the 1970s, all organisational calculations, including scientific computations, data processing, data input, word processing, and so forth, were handled by the primary computers. As the personal computer didn't exist until ten years later, using such an expensive machine—sometimes costing millions of dollars was the only option. Even though they were expensive, these computers weren't particularly effective at carrying out all of these different kinds of calculations. Nevertheless, a number of businesses initiated a

differentiation process with the purpose of creating computers that were appropriate for certain calculations, such as scientific, data processing, and so forth. Even major corporations found it difficult to afford more than one computer due to the still-high cost of such a machine. All calculations have to be managed by the current computer. Their inefficiency sharply rose when more computers began to provide interactive features. The very earliest computers were "dumb" terminals, incapable of doing any computations. The primary computer received each character that was input. The computer had to momentarily pause the current application, launch a tiny system tool to validate the character typed, and then send a command to the terminal to display or print it if it was valid. An error notice had to be shown if the character was invalid. The computer had to pause the application it was executing since the user would have noticed a greater delay, perhaps 200–300 milliseconds. The original application that was executing when the character was received would not resume its execution until the system utility had completed processing the character.

Computers of the 1980s: The Last Mainframes

Organizations' perceptions of their computing environment evolved as a result of the quick advancement of computing technology, the "birth" of personal computers, and their ever-growing capabilities. The new developing platforms offered some improved capability and were substantially less expensive than the computers used by major organizations. A simple price/performance analysis was sufficient for certain forms of computing to replace the operating system. For instance, running a simple word-processing application on a central computer would not only be inefficient, but also costly and time-consuming. Using a dedicated PC is far quicker and much less expensive than the mainframe's time-consuming procedure of interrupting the active application for each character input.

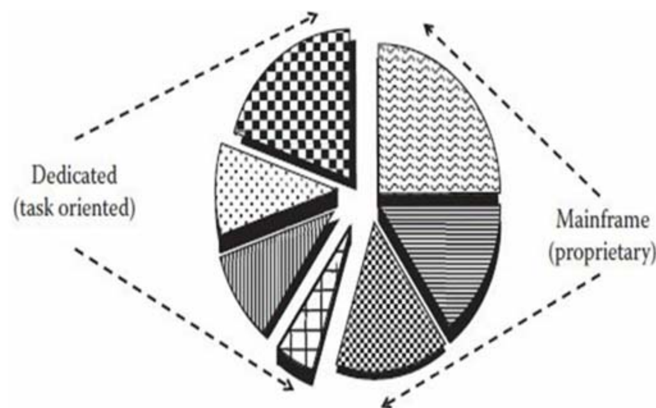

Figure 6.1: Illustrates the Changes in computing environments.

The mainframes continued to focus solely on a few kinds of computing as a consequence of the trend towards identifying the optimal platform for each type, while other types were shifted to specialized and task-oriented computers. The conventional mainframes and the kinds of calculations that are still used are right side. The kinds that were moved to other, more effective and less expensive platforms are shown on the left side. Due to their various capabilities and the fact that most mainframes were proprietary platforms whereas most PCs and subsequent servers were off-the-shelf products, the systems' pricing varied greatly. The majority of the time, several manufacturers produced these off-the-shelf products in enormous numbers, and they were substantially less expensive.

It should be noted that significant advancements in networking technology were necessary to enable these distributed designs, in which certain sorts of computations are conducted on a

range of platforms. Further functionality was needed in addition to the previously mentioned advances, such as the automated transfer of work based on their nature. Since computer platforms are dynamic, users cannot manually conduct these transfers because one machine can be replaced by another. The users should have complete transparency about these substitutions (Figure 6.1).

New and more dispersed designs were made possible by all these new network functions that routed tasks to the right systems, returned the results to the user, or printed them on the nearest printer.

"The Computer Is the Network": Scott McNealy, one of the company's founders who subsequently served as CEO until Sun Microsystems was purchased by Oracle Corporation in January 2010 during the 1980s, was a proponent of a new computer system based on a networked environment at the time. Many attacked the proposal at the time, saying Sun Microsystems was trying to get around its constraints. Sun Microsystems, a new computer company, focused on tiny and medium-sized computers, particularly inexpensive ones. Unlike the specialised and very costly cooling systems employed by the mainframes of the day, the environment utilised to operate these computers was a typical air-conditioned room. Whether McNealy was attempting to provide a high-end solution based on Sun's platforms or whether he really believed in the new trend is unclear, but he was able to predict the future with accuracy and most likely contributed to moving in that direction. The primary concept embodied by the networked environment is the comprehension that computing develops into an integrated ecosystem in which resources and computers are linked invisibly. The computer user is unaware of and unconcerned with the particular system that powers the program they are using. A networked environment where several heterogeneous systems work together to give the necessary solution provides the service. Sun created the Java programming language, which offers a universal architecture capable of executing standard code on a number of systems, to support this concept. These platforms include numerous devices that aren't often thought of as computers (e.g., TV sets). A very cautious estimate states that there are many billions of appliances that can run Java.

Sun Microsystems' concept is an additional evolution of a networked environment, which cleared the path for the contemporary, globally linked world as we know it. Sun's concept is very clearly represented by today's straightforward browsing. We obtain a variety of computing services from several systems located all over the globe when we are surfing. Most of the time, we don't even know the system's location, make or model, operating system, or other details. No one really cares even if there is a means to learn more about the server that offers the service. All necessary migration is handled by a fundamental network component, ensuring that the answer we obtain is understandable and comprehensible. The usage of the cellular telephone network may be used as a straightforward parallel for the necessary migration. Everyone has a mobile phone that can communicate with every other device on the network, not simply those that are comparable to them or that use the same network protocols. In this instance, the network is in charge of managing all the migrations required to link all these devices. A network of computers is providing organisational computing services to a user. These services are sometimes carried out on the user's local computer and occasionally on a close-by server. Other times, it could be carried out on a distant organisational server, and sometimes, it might even be carried out on a system outside the organisation that has no connection to it at all (e.g., search services). The majority of the time, the user is unaware of the service's origin. It should be emphasized that this infiltration affects a wide range of devices in addition to computers, including mobile phones, Internet cameras, GPS units, and many more appliances.

Computers in a network

Parallel to the PC's quick entry into the computer world in the 1990s, a select few businesses controlled the market, experienced economic success, and saw significant growth in their customer base. Microsoft, which sold the majority of the PC software, and Intel, which was the main provider of hardware components, were the two key participants. Following market trends is necessary if one wants to comprehend how the market behaves. Microsoft updates both its widely used Office suite and its operating system every couple of years. Sometimes this entails significant adjustments that even need rewriting some of the code. There is a need to update the software because of how linked the globe is and since most items are not upwardly compatible. Even when security risks are implicated, Microsoft does not update outdated products or impose maintenance costs for marketing reasons. Regrettably, this implies that consumers must buy the new updated product since Microsoft does not provide an upgrade for the new product. Assume, for instance, that a user gets a Word, Excel, or PowerPoint file as an attachment to a mail message. There is a potential that the receiving user won't be able to access this new file if it was made by a new version, particularly if he or she is using an earlier version. This implies that occasionally events outside of the Computer affect how the programme is updated. As a result of the considerable and quick advancements in PC technology throughout the 1980s, many software updates also called for hardware upgrades or, less often, the purchase of a new Computer. For the house or an enterprise to maintain computer capabilities, this manner of operation needed significant financial outlays. It should be emphasised that many other specialised businesses were able to participate in the expanding market and benefited from these methods of making repeat purchases. Some businesses, on the other hand, didn't take part in this activity and were seeking for a strategy to grow their market share concurrently with or in place of the market leaders.

Five computer firms (Apple, Oracle, IBM, Netscape, and Sun) created a new partnership and coined the term "network computer" in May 1996. This groundbreaking notion at the time was based on a number of presumptions:

Connected computers: The majority of personal computers are linked to one or perhaps many networks. It may be a personal or business network at times, and the Internet or another wide-area network is often used.

The network is no longer a bottleneck: Fast data transfers were made possible by recent networking technology advancements. Even if the transfer rates were still mild at the time the new idea was announced, it was realistic to expect that they would substantially alter. In hindsight, this was a sensible and correct assumption. In comparison to earlier technologies, the ADSL technology, which enabled home computers to connect to networks via a telephone line, was inexpensive and substantially quicker.

The Microsoft model shows how the cost rise brought on by computer interconnection. For instance, the advent of Office XML formats resulted in several new Office editions changing the file format. This indicated that an update was necessary in order to access these files. Unfortunately, an update was often not an option, therefore the customer was forced to buy a new licence. The PC sector, which was once controlled by the Microsoft infrastructure, was the only one to experience this cycle of constant spending merely to maintain operations. Total cost of ownership, or TCO, was coined in the 1980s by Gartner, a global leader in IT consulting, as a way to describe all expenses related to computing across its entire life cycle, including acquisition, use, maintenance, and decommissioning. The Computer itself costs about $1,000, but according to research that was released at the time, the true cost (which includes administration, support, maintenance, backups, updates, infrastructure, etc.) may be

as high as $10,000 year.

Making Use of Characteristics

We must discover the computer qualities that use the majority of the budget in order to better comprehend the circumstances behind these excessive prices. The system may be conceptualised as the integration of six parts, including the CPU, memory, storage, display, keyboard, and mouse. According to an analysis of PC upgrades, the majority of changes have been made to the computational components (CPU, memory, and storage), while display resources have essentially not changed over time. It should be emphasised that while the technology of displays has advanced from a monochrome to a sizable colour screen with greater resolution, this advancement pales in comparison to the advancements made in the processing components. The functioning of the mouse has not changed over the last 30 years; it is still used to point at and select data on the screen. Instead of mechanical mice, we now use ergonometric, optical, and wireless mice with better resolution. Almost else has stayed the same, including the keyboard. The essential operation of the new keyboards remains the same, albeit they may contain extra buttons for managing the system or specific programmes. Humans' limited ability to utilise these gadgets meant that they did not need to be any quicker. On the other hand, computing resources have seen a tremendous improvement in performance. Between 1982 and 2010, PC processing power rose by more than four orders of magnitude (from a 4.77 MHz 8080 CPU to the latest 3.0 GHz multiple core devices). The internal memory of the PC also grew, going from 640 KB to the contemporary 4 and 8 GB*, by a factor of more than five orders of magnitude. Nonetheless, the amount of external storage expanded significantly, going from floppy discs with 360 KB to contemporary drives with over 2 TB.

the adjustments in computer power as programme functionality grows. The computer resources typically see a large growth, although the display resources seldom ever do. Users often update their keyboards and mouse when purchasing a new desktop computer since the expense of doing so is not prohibitive. The cooperation between the five computer businesses drew a comparison between computing behaviour and that of the telephone or television. In these two instances, the user-side hardware is straightforward (similar to the display resources), therefore regular updates are not required. The facility (telephone exchange or TV transmitting station) is equipped with robust and strong machinery that can handle the responsibilities. The difference between the two is obvious, and improving one side has no impact on the other. Yet, it should be emphasised that the twenty-first century is a little different. The reason why the present generation changes or upgrades their mobile phones isn't because they've broken down, but rather because of aggressive and effective marketing tactics.

The notion of a network computer was inspired by the distinct distinction between the client side and the central service provider. It was planned for the network computer to be a "thin" computer system that relies on the network to operate. Since all the data is kept on a networked server someplace, such a system does not need a storage device. This data will be loaded over the network if and when it is required. Additionally, the system may be started by downloading the required software from the network before it even boots up. Each of these network computers has relatively little processing power but the necessary display resources (screen, keyboard, and mouse). The network computer runs some of the programmes locally using its own memory and processing power, in contrast to the dumb terminals (see the section "Historic Perspective" in this chapter). Yet, compared to conventional (non-network) computers, the computational resources are somewhat constrained. A reduced TCO results from the network computers' ability to perform comparable functions without the need for

periodic upgrades. Also, since these computers did not come with storage capabilities at first, they offered a stronger and more secure computing environment that was impervious to virus attacks and other nefarious efforts. The initial network computer concept also included the significant maintenance expenses. Any modification made to the programme is carried out on the master copy, and the network computers will always use the most recent version when the application is required. A remote Computer may presently be loaded or upgraded in a number of methods without requiring physical access, it should be highlighted. Unfortunately, when this concept was proposed, there were restrictions on remote access, so information technology (IT) personnel had to physically visit each system to install the updated version. User engagement was another expense component that increased the TCO of conventional PCs. Several times, users wasted hours of production time attempting to fix different computing-related technical issues. This was brought on by both their lack of expertise in handling the issue and the fact that they weren't carrying out the jobs for which they had been engaged. Users using network computers have limited options for issue solving since everything is controlled centrally.

Many people were overwhelmed in the early years after the notion was presented, and several publications and studies with estimations that network computing would reduce the TCO by at least 30% were released. International Data Corporation (IDC), a consulting company, anticipated that by 2005, network computers will make about half of all computer shipments. In hindsight, this did not occur.

The network computer concept needed a new infrastructure of renting software in order to be implemented successfully. As network PCs lacked storage capacities, it was impossible to store any applications or data files locally. The goal was to create a new market for service providers where businesses would rent out software. With network computers, the notion was that a user would rent the software for a certain amount of time as opposed to the typical approach in which the consumer purchases a software package. Other utilities, like telephone or electricity, were the inspiration for this concept. The user signs up for the service but only pays for what they really utilise. While analogous marketplaces, like those for ISPs (internet service providers), previously existed, the dominance of Microsoft in the PC software market prevented the implementation of such a model without Microsoft's direct involvement. Microsoft chose not to take part because it was aware of the possible harm that such a model may do to its company. Microsoft's business strategy at the time was centred on repeat business. The majority of customers at the time were using Windows 98, and it was expected that they would upgrade to Windows 2000 and then again to Windows XP in a few years. The customers really purchased a new licence even though this was advertised as an update (sometimes with a price reduction). If network computers are adopted, fewer licences may be purchased, which might reduce Microsoft's income. Despite the fact that network computers did not develop as numerous observers had predicted, several more measures to reduce the expenses related to personal computing arose. One path evolved into the open-source development approach, which makes free software alternatives available to a larger audience. However, several businesses, like Citrix, were created and provided a variety of terminal-based solutions, partly realising the concept of network PCs. It's even possible that the concept of network computers served as some of the inspiration for the current trend of cloud computing, which will be covered later.

Services at Terminals

Microsoft began developing a notion that was comparable to the network computer idea but would guarantee the future cash stream concurrently with its efforts to block the idea. Microsoft's somewhat successful response to network computers was the Terminal Services.

These are a group of software programmes that enable terminals (PCs with Windows operating systems) to access programmes and information kept on a centralised server. The multiple advancements underlying the Terminal Services technology aimed to reduce costs and increase efficiency. Time and money were saved since all of the programmes were deployed, managed, and maintained on a single server. The amount of time saved was much more substantial for bigger enterprises. In addition to taking a lot of time, installing a programme on thousands of PCs involves a difficult logistical process that is sometimes impossible, particularly if the systems are spread out geographically and different versions demand for simultaneous installs.

It is less expensive and more effective to support and manage the apps from a single place, which results in further organisational savings for the IT department, lower downtime, and more productive users. The execution environment also causes a difference between the two architectures. One of the primary causes of the ongoing need for upgrading on a regular PC is the programme running on the machine. With a network computer, however, certain functionality may be carried out remotely on the server. This implies that the client's need for computer resources is low, resulting in a longer interval between hardware upgrades. As the majority of the resources are located on the server side, even outdated and rather sluggish PCs may be used as network computers. The initial concept of network computers received an extra layer of capability from the Microsoft implementation. The system included a number of parts that could be installed on non-network PCs as well as Windows-based systems.

Also, after the Terminal Services were established, Microsoft ceased being against the concept of network PCs. This only occurred when Microsoft modified its licencing policy to require customers to buy licences for each linked PC, even in network computer arrangements. The primary goal of network computers, which was to reduce the price of computing, was obviously compromised by this. Microsoft started presenting a remedy like to the one suggested by the network computers only after the new licencing system that blocked new participants from joining the software rental market vanished. Nonetheless, it should be mentioned that Microsoft recognised the market trends and began to provide a range of price reductions, such as site licences or volume discounts.

The notion of network computers was implemented by Microsoft, and this gave the company the tools to accomplish an extra strategic goal. Microsoft has been vying to become the standard desktop computing option in businesses for a while. Microsoft operating systems (Windows XP, Windows 2000, Windows Vista, Windows 98, Windows 2003, and Windows ME) were discovered on 94.2% of the computers evaluated, according to a report released in May 20077 by Awio Web Services. This indicates that at the time, Microsoft led the desktop market, followed by Apple with 4.4% of the market. Regrettably, a different image emerged from the same study in September 20148. Despite the fact that Microsoft remains the key and preeminent company, its operating systems were only present on 61.4% of desktop computers, which is a considerable fall.

Client/Server

Client/server technology is a crucial component of both Terminal Services and network computers. Client/server technology underwent multiple configurations as it progressed through time, much like other computer technologies. The mainframe and the emergence of organisational networks were the primary drivers in the early phases (for more information, see this chapter's section on computer networks). The technique was created to lessen the pressure on very costly corporate computer resources and to more effectively exploit the PC's inherent capabilities. Client/server technology essentially aims to enable different means of

task sharing between the client side (the desktop Computer) and the server side (the organisational mainframe).

Data Server

Initially, file sharing was the principal purpose of PC networks. The design was straightforward, with a server serving as the main location for file storage and multiple personal PCs sharing a network. Each time data was required, the whole file was sent from the server while the programme was running on the client (the local personal computer). This manner of operation worked best when there were few computers connected, few data were exchanged, and the files' sizes were reasonable. The need for a locking mechanism to ensure that no other computers may access or edit a file while it is being processed by one machine presents an additional issue with this sort of implementation. It was necessary to modify the operating system so that it could monitor the files and their use as well as provide tools for resolving tricky circumstances.

Think about a scenario where a user was working on a file but forgot to release it before leaving for the day. It may take some time for the file to become accessible, therefore all other users who need access to it are placed on hold until then. This indicates that, although being an excellent notion for sharing methods, file server technology has certain drawbacks. It is helpful when there are few linked personal computers and not many huge files are shared, but even then there might be an error scenario. The initial file server architecture implementations aimed to get around another Computer constraint. The initial PCs only had a small amount of storage, but the server had reliable, high-volume storage capacity. An organisation was able to keep a single, current copy of organisational data in this manner. Also, creating backup plans for data security was simpler.

A more adaptable architecture was created as a result of the inherent restrictions of the file servers' initial design, particularly the need to send the whole file each time a piece of information was needed. On the local computer, the files on the server may be "mounted," making them all appear as a brand-new "virtual" drive. This meant that the programme could read just the data it needed rather than having to send the whole file via the network, and that the file was really handled by the local operating system (with some necessary changes).

Database application implementations have a similar design. Each local personal computer runs a copy of the database management system (DBMS) software, while the database (the data) is stored on the server. While the data is stored remotely, the processing is local (on the server). The whole file is sent in the manner indicated each time the programme wants to access or edit the data.

Client/Server

The more sophisticated client/server design was born out of the inherent constraints of the file server architecture. The file server was swapped out in this implementation for the database server. While interacting with a relational database, the application sends the query and receives the response from the DBMS. In this instance, the DBMS accesses the database, extracts the necessary data, and then provides it back to the client. The volume of data exchanged is significantly decreased in this mode of operation. Just the targeted record (or records) are being sent back, not the whole file.

A client/server design divides processing between the client and the server, in contrast to the file server architecture, which mostly uses the local personal computer. It should be noted that while the design was first used in local area networks (LANs), it may also be used to

wide area networks (WANs). All of the customers are aware of the services the server offers. This indicates that while the client is aware of the server, the server is not required to be aware of the clients.

Client/server design has advantages but also has certain disadvantages, like any other technology. The primary advantages include

Effortless use: Distributed application development is simple and practical (including distributed data). Even in separate places, the data may be stored on distant servers. Even when services are moved from one server to another as a result of server consolidation or the addition of new ones, the behaviour of the local system remains unaffected.

Resource management: The network is not overwhelmed with unnecessary file transfers. The local system (the client) and the distant system each run different applications (the server). This method could stop needless and expensive hardware upgrades. When an update is required, it may be carried out by seamlessly moving services to new hardware without interfering with the user's work.

Client/server architecture's disadvantages result from its capabilities (a distributed system in which the data is sometimes not stored in one central location).

It is challenging to develop a single model of the whole organisational data when it is scattered over numerous (or even many) sites. Integrity problems as well as pointless duplications might result from this. In certain circumstances, more effort may be required to manage the data resources. It's conceivable that customers are unaware of all offerings. Such circumstances call for an extra service. Service discovery will be handled by the new service.

It should be emphasised, nevertheless, that the aforementioned shortcomings are more directly attributable to a poor implementation of the client/server design than to the architecture itself. The provision of pertinent information to the different execution levels for improved decision-making is one of the key concerns in the implementation of an information system. In certain cases, this means that managers must have access to all corporate assets, including data.

Under these circumstances, a poor client/server architecture implementation might let part of the data to be stored on local Computers. This problem reveals itself in the effectiveness of the decision-making process since the data is often kept secret from other users and management and is typically known only to the specific user. There are a number of ways to construct a client/server architecture, but in each case, a portion of the calculations are handled by the local Computer. This design is based on the notion that a PC is a standalone computer that can be used to offload the main computer rather than merely replacing dumb terminals (see the chapter's section on "Personal Computers" for more information). However, certain tasks, such handling input and output, are better handled by the local PC because of its close proximity to the user.

There are two primary options for setting the desktop computer (the client) in a client/server architecture: Thin customer, who hardly participates in the process. The server handles the majority of processing and storage management, with the client merely handling data input and presentation. Customers that had used mainframes for a long time and were used to the central computer being the dominating component of the system favoured this arrangement. In these scenarios, the server (which takes the position of the mainframe) continues to handle the majority of the work, with the clients playing a mostly supporting role. To guarantee that this implementation's server can handle the necessary workload and avoid becoming a bottleneck, it must be strong. The thin clients are old-style dumb terminals with a few more features.

A more powerful computer known as a "thick client" conducts a significant portion of the processing. The database is controlled by the server (which serves as the organisational storage resource), while all other processing and presentation is carried out locally. As most software is kept locally and updates are more complicated and time-consuming, this option proved more challenging to handle. This was, of course, just the beginning since, as in many other instances, a number of solutions were created to remedy this issue, and at the moment remote installs are the primary method of maintaining these sometimes faraway systems.

CHAPTER 7

IMPLEMENTING SERVER ARCHITECTURE

Gaurav Londhe, Associate Professor,
Department of Computer Science and Engineering,
Jain (Deemed to be University) Bangalore, Karnataka, India
Email Id- gaurav.londhe@jainuniversity.ac.in

A two-tiered structure the local client and the distant server are the two machines involved. The processing is divided between the two, as with all client/server designs, depending on the kind of client (thin or thick) or how the programme was created. The DMBS and other operations that can be carried out remotely and make the creation of applications simpler, including stored procedures, are often located on the server. When there are just a few people using the server, the two-tier (or two levels) design offers a viable option. If there are plenty of users, the response time can dramatically decline. This is because the server keeps lines of communication open with each client, even when there is no work to be done. Also, since it is difficult to divide services or add more servers in situations where it is predicted that the number of clients will rise, it is best to avoid using this design.

a three-tiered structure This was created to get around the two-tier architecture's constraints. Between the client and the server, a new layer, or tier, was created in this design. The implementation of this additional layer may be done in a variety of methods, such as by adding a messages server, application servers, transaction monitoring, and so on. Queuing services may be provided by the central layer; when a client submits a request, the queuing server adds it to the queue and makes sure it is processed when the appropriate moment comes. This indicates that the customer was relieved of responsibility for ensuring its operation. For managing a large number of clients, this design is helpful since it is adaptable and can take modifications as well as the addition of additional servers. On the other side, more work is required in this instance to construct the application. The three-tier architecture's adaptable and modular design makes it possible to include outdated servers that store significant amounts of valuable data into a contemporary solution. The three tiers (or levels) are sometimes referred to by other names, particularly throughout the software development process, although the idea is always the same. The presentation layer, which is the top layer, is in charge of user communication (input from the user and display of the results). The local PC implements this layer. The application is the second layer (or the logic to be performed as part of the application). Sometimes, as with thin clients, this layer is implemented on the server; other times, as with thick clients, it is implemented on the client. Even splitting this layer between the two systems is possible (client and server). The data layer, which is the third layer, is implemented on the server.

The definitions and ways in which they are implemented have evolved and continue to change due to the fast growth of technology. The development of two- and three-tier designs opened the door for further tiers to be added if necessary, such as four-tier architecture, which contains an extra layer or even a new manner to divide functionality into layers. The Sun Microsystems-created J2EE architecture, which is now a part of Oracle Corporation, is a multitier architecture that implements the client tier and the functionality it offers: the Web tier offers the tools required for web-based operations; the Business tier offers the solution for the particular business segment; and the EIS (Enterprise Information System) tier offers the

database. Many physical computers may be employed to implement these logical layers depending on the infrastructure in use.

The extra layer may be used to add new layers or to provide additional services as a part of this layer. The TP monitor (transaction processing monitor), which maintains transaction queues, organises work, and offers a way for adjusting the priority of different transactions, is a very basic but crucial function. TP monitor technology had previously been developed a number of decades before as a result of the need to synchronise several users (clients) operating on a single system. A TP monitor is required because to the widespread usage of web-based systems where the number of concurrent users is unknown yet the system must be able to handle all of these requests. As they act as a conduit between the systems that supply the different services that are requested by clients and those systems, the software components that support distributed processing are frequently referred to as middleware. This technology can control many databases simultaneously in addition to managing the task. Link to various data sources, such as non-relational databases, flat files, and data stored on proprietary systems or other non-standard platforms. Offer tools for establishing transaction priorities.

Improve and expand the security infrastructure: The presentation layer, the business-logic layer, the DBMS layer, and the data layer are the four core application components that are used to illustrate the many designs that might be utilized.

A conceptual representation of the mainframe architecture. The presentation layer was processed on the server side since the dumb terminal was incapable of doing any computations. Nevertheless, it should be mentioned that this paradigm was used in the 1970s and early 1980s before the concept of layering applications was invented. Nevertheless, the scenario in which the mainframe handled every aspect of execution while the client just offered straightforward data input and output services, printing the result first before displaying it afterwards. This is precisely the network computer paradigm that the original five computing businesses had in mind (for more information, see this chapter's section on network computers). The local computer does not need a fast processing unit and does not require a lot of storage space since it just serves as the presentation layer and handles input and output. Applications, data storage, and all business logic are retained on the server side.

In this instance, the programme is divided between the client and the server, with certain functions being handled locally and others being handled remotely. The server houses both the DBMS and the data. Thin clients are no longer an option for supporting this architecture since the client is now responsible for operating a portion of the application and must thus have the necessary infrastructure. In order to store the temporary data it processes, the client may need some storage capacity. The particular architecture will be created as part of the system analysis and design, which will have an impact on the architecture and implementation. It may be challenging to execute modifications that involve shifting some functionality from the server to the client after the architecture has been established and the application has been created. It should be noted that the recommended network computer model also accommodates this kind of architecture.

Many devices that only partly implemented the concept of network computers have been developed since the network computer notion was originally proposed. By utilising numerous thin client types throughout the course of several years, the following broad knowledge was gained: Compared to a typical Computer, a thin client requires far less setup. The growing difference between network terminals and regular PCs, on the other hand, is a more intriguing topic. This is mostly caused by the rise in PC hardware specifications that comes along with new Windows versions. Working on a network terminal is identical to working on a regular

Computer. The user often has no idea what particular setup they are utilising. With a comparable windows structure, the functioning is extremely similar to the Windows-based operating system. The network terminal only needs a small amount of physical memory, but given current memory cost trends, thin clients nowadays use typical amounts of memory (2–4 GB). Compared to ordinary PCs, the total cost of ownership is cheaper (often noticeably lower). The original arrangement is not only easier to maintain, but is also simpler.

The thin client, which adopted a new way of operation involving remote processing, offers new features adapted from other operating systems like UNIX. A lengthy process may be started by the user, disconnected from the server, and allowed to run on its own. The system will remember that the process is still active (or waiting) when the user reconnects, and it will offer to re-engage the user with it. Thin clients provide greater malware protection since there is no local storage involved. Nevertheless, it should be emphasised that the server is still susceptible to attacks from viruses, for instance, even if the servers are often run by IT department experts who follow stricter security protocols. Thin clients provide extended use of the computer (as was the main idea represented in the network terminal). As a consequence, there is less of a need for updates, which brings down the price of computing. The network terminal represented a solid concept, but it was regrettably premature at the time, according to the collected experience. Further "Computers" Throughout the years, special-purpose computing devices were developed in addition to the "ordinary" computers that have been covered in this chapter. These are several gadgets that sometimes have a particular feature for a certain requirement. Nonetheless, because of the value they provide, some of these devices are supplemented in the corporate network. The handheld gadget, including smart phones, is a well-known and prominent example. Even if their initial purposes were often diverse, several of these gadgets had a significant positive impact on the organisation. For instance, they provide online and sometimes real-time updates even when the individual updating is not present in the office. The widespread usage of computers in household goods opens up many possibilities for cutting-edge inventions like elevators or autos that may identify issues before a user even realises or is negatively impacted. All of the embedded processors that are found in the many tools and appliances that we use every day are essentially computers that may be used in a number of creative ways. The notion that the world would eventually become a vast system of linked devices was one that informed the creation of the Java programming language (see the section "The Network is the Computer" in this chapter). It is known as the "Network of Things" or the "Internet of Things" in the first decade of the twenty-first century. It is a vast system of interconnected "things" that work together in both instances. These "things" might be anything tangible in our life (clothes, glasses, home appliances, etc.). They will all be programmable and function as separate entities inside the network. With the use of technology, some of the "things" will be able to be "programmed" or have their functioning altered. Although while this may seem like science fiction, one should consider that if concepts presented twenty years ago had the capability of modern mobile phones, the reaction may have been comparable. But, at the moment, software developers may create unique features that can be installed into their phone, enhancing its usefulness, using SDK (software development kits) offered by manufacturers.

In actuality, people in the twenty-first century are surrounded by a variety of systems, often to an extent that we are oblivious of. As many of these systems are computer-based, they provide all the capabilities that a computer has to offer. Just a small portion of them include electronic games, watches, air conditioning, music players, cameras, traffic signals, and medical devices. These "things" often need to interact with a hub or a bigger server that oversees and governs their functioning. The long-standing server idea has seen major

alteration in recent years. Several major corporations began to provide an alternative computer model that is focused on service after realising the high costs of computing. This approach, "Other Architectures," is known as "cloud computing," and it is essentially a client/server architectural variant in which the servers are managed by several big businesses and are located far away. The Internet is used for the connection. The cost savings is one of the main advantages. In contrast to the "normal" computing model, which requires the company to buy all the equipment required to handle peak demands, the customer just pays for the services they actually use. Such peaks may be quite high for Internet-based online systems, necessitating a big and sophisticated setup that is seldom employed. The provider in the cloud computing model has a very big configuration that can handle even the highest peaks, and this configuration is accessible to the clients anytime they need it. Other outsourcing possibilities offered by the cloud computing architecture, such as the ability to scale down on both space and IT workers, translate into further cost reductions.

Interactive people may connect with a computer via a terminal, which is a device. A keyboard and a printer are often used to input and display data, respectively. A screen that shows the data and graphics has taken the role of the printing device in current terminals. The act of signalling the computer to halt and let it handle an external problem is known as interrupting. The currently operating application is momentarily halted or stopped to handle a more urgent situation. A new computing architecture was described by the phrase "network computer," which first appeared in the 1990s. It should be emphasised, however, that it has nothing to do with networked computers, which are often just a fancy way of saying connected computers. While a network computer must be linked to a network, it has very specific characteristics that set it apart from networked PCs.

High transfer rates are made possible by the quick data connection technology known as asymmetric digital subscriber line (ADSL). The number 109 is the giga (bytes) symbol. The first action a computer does after being turned on is known as booting up. A preliminary hardware test is often conducted first, and then a short piece of software that will load the whole operating system is loaded. The phrase was first linked to the well-known Rudolf Erich Raspe tale of Baron Munchausen's antics, in which the Baron pulls himself out of the swamp by pulling his hair. A database is a structured collection of data that offers a simple and effective method for storing and retrieving data. The information gathered from the database and the data kept within often give a picture of the reality in which the organisation operates.

The database management system, or DBMS, is a piece of software. Its responsibilities include taking data, classifying it for storage, and retrieving it as required. Between the real tables and the programmes (and users) accessing the data, the DBMS may be thought of as a software layer. It deals with the data on the one hand while interacting with the data consumers (users and apps) on the other (defines, manipulates, retrieves, and manages). An advanced kind of database is a relational database. The first databases employed flat files, where each entry was separated from the next by a unique character in a large text file. On the other hand, the relational database is built on tables, each of which represents a different sort of information (customers, products, employees, etc.). Each row in a table represents a separate entity, and each column in a table represents an attribute (or field in a record, such as a customer's name, phone number, or address). Because of the relationships that are kept, it is known as a relational database. Although certain columns in some tables may be used as keys, other tables provide the option of building new tables from the ones that already exist. Without the need to replicate the data, these links boost the database's flexibility and effectiveness.

A local area network (LAN) is a network (hardware and software components that provide networking capabilities) for a collection of systems that are physically adjacent to one another, such as a house, a working group, a department, a building, etc. A network that covers a vast geographic region is called a wide area network (WAN). It often contains many LANs and offers a way to connect to a wide-area telecommunications network. A platform for the creation and deployment of corporate Java applications is called Java 2 Platform Enterprise Edition (J2EE). The platform improves on the earlier standard edition (SE), adding new layers for greater dependability and sustainability.

Representation of Data

With the extensive use of computers and computer-based systems, a clear definition of data representation is necessary. Even if most users are unconcerned with the underlying representation of the system and human-computer interaction is at a high level, it is necessary to guarantee the system's correct operation. This notion is comparable to "protocols" that were developed to facilitate communication among people, such as natural languages. In attempt to provide a way to convey words in a more visual way, writing was created. A collection of symbols (letters and numerals) that stand in for the language's established sounds are used to do this. The development of writing symbols (letters, numbers) and the resulting creation of books, newspapers, and the information shown and printed by computers were only possible after the definition of languages. The mechanism for written communication that is not limited to face-to-face conversations was formed in the early phases of human development using the standard for portraying a natural language. Ancient Egyptian hieroglyphs and the almost 5000-year-old Cuneiform writing are two very well-known examples. The ability to speak with individuals even when they are far away is made possible by the development of analogue and then digital communication lines and the rapid evolution of technology. The different technologies (such as telephone, telegraph, fax, etc.) required to employ a preset encoding scheme in order to create such connections. The Hollerith punched card system, which employed the holes in the card to represent data, was one such system that has previously been noted.

Data representation standards needed extra care due to the Internet's rapid expansion and global system status. The fundamental framework for data flows between all linked devices is provided by these standards. Additionally, since the binary system is used by all contemporary computers, the binary representation of data must also be defined by the standards. In this data, there may be text, special symbols, and numbers (integers, real, and complex numbers). The capacity of the representation system for numbers to allow calculations is a crucial component, as will be discussed in the chapter's section on computer arithmetic.

Computerized Systems

Humans have need a technique for measuring things from the beginning of time. As a result, a method to categorise size was devised as part of fundamental verbal communication. Several similar numerical systems, first used for counting and subsequently for calculations, have been evolved throughout time by diverse cultures. Such numerical systems are necessary for data representation since the system must be established first and its representation determined only subsequently. Quantities must be represented by symbols (numerals) in any numerical system. The quantity of symbols employed in the numeric system must be kept in a careful balance. On the one hand, there shouldn't be too many symbols, making it simpler for people to memorise and apply the system. On the other hand, it shouldn't be too tiny since this would call for a lengthy representation (or several digits, as will be covered in more detail and discussed in this chapter's section on the binary system).

Of course, the system must be capable of handling the whole range of numbers (i.e., be infinite).

These ancient number systems are supported by a large body of archaeological data. These methods were first designed for measuring and counting things like the size of a herd, the number of persons, etc. The hieroglyphics system was expanded by the ancient Egyptians to incorporate numerals. The system was based on a number of symbols, each of which stood for a different value. Each number in the system was described by its constituents and was based on decimal (base 10) numerals. Every power of ten has a unique symbol. The values of the fundamental numbers (symbols) were 1, 10, 100, 1000, 10,000, 100,000, and 1,000,000.

In general, the value is determined by adding up all the numbers; however, certain values are determined using subtraction in order to streamline the computation and reduce the number of times the numbers are used. For instance, the Romans defined four as five less one, but the Egyptians symbolised the value of four by four lines (repeating the number one four times). According to the reasoning used, all numbers are written consecutively, starting with the largest and working down to the smallest. The left number must be deducted, nevertheless, if it is smaller than the one after it. For instance, the value of 2014 is represented by the Roman numeral MMXIV:

$$\mathbf{M + M + X + (V - I) = 1000 + 1000 + 10 + (5 - 1) = 2014}$$

Similarly, MCMXLIV represents the value of 1944:

$$\mathbf{M + (M - C) + (L - X) + (V - I) \ = \ 1000 + (1000 - 100) + (50 - 10) + (5 - 1)}$$

Decimal Numbering System

The currently most widely used numbering system is the decimal (derived from the Greek word deca, which means 10) system, which is based on 10 numerals (digits), each one representing a specific value. A number is written as a list of digits wherein each location corresponds to the specific power of 10 multiplied by the digit in that location.

For example, the number 1987 is calculated as 7 * 100+ 8 * 101+ 9 * 102+ 1 * 103 The general migration formula is

Further Numerical Systems

While frequently used, the decimal system is just one example. The formula previously stated may be applied with any number of other systems, therefore there are likely to be many more that can be employed. It is sufficient to write the number when the base is known; however, if it is unclear which base should be used, the base is appended as a subscript digit to the number. For instance, the value of the number 573 when written in base 8 is 5738. As it is one of the bases that is a power of 2, base 8 or octal, which is derived from the Greek word octo, which meaning 8, is a significant base in relation to computers. In a similar manner, the value of 5738 is computed:

$$\mathbf{573_8 + 3 * 8^0 + 7 * 8^1 + 5 * 8^2 = 3 + 56 + 320 = 379_{10}}$$

The digits that are accessible in the octal system are [0:7], just as in the preceding example. Mathematical computations may be done in any basis; however, because humans are used to working with decimal numbers, translating numbers that are expressed in other bases into decimal numbers is necessary to determine their values.

No matter what base a number is defined in, determining its value is always the same. For example, the value of 43215 is 58610:

$$43215 = 1 * +5^0 + 2 * 5^1 + 3 * 5^2 + 4 * 5^3 = 1 + 10 + 75 + 500 = 586_{10}$$

Binary System

The binary base, 2, is a highly significant base, particularly when addressing computers. Despite the fact that the original computers used decimal numbers, this swiftly changed after it was realised that computers, like any other electrical circuit, can only function in two fundamental states: on and off. Due to this insight, computers are now built using contemporary architecture that is based on the binary system. The binary system only has two numerals, but other than that, there are no differences, and determining the value of a particular number is the same as with other bases. In the binary system, the digits are known as bits (binary digit).

While the binary system is effective and handy for computers, it poses a considerable challenge for people. In order for a numerical system to be readily recalled by humans, it should have a minimal number of digits, as was previously mentioned (see the section "Numerical Systems" in this chapter). On the other hand, this number (of digits) shouldn't be too low since a number would then need to be represented by a lot of digits. The binary system only has two digits, therefore huge numbers are represented by a large number of digits.

As a result, eight bits are required to represent a number with three decimal digits (or eight binary digits). When the number is much higher, the number of bits quickly rises, making it challenging and sometimes impossible for humans to follow.

Using Actual Numbers to Represent

The representation of natural (integer) numbers was covered in earlier chapters, but real numbers constitute a far bigger class of numbers. Due to the fact that fractions may be described as digits multiplied by negative powers of the base, the logic and explanation of this group are extremely similar. Real numbers may be migrated using the prior migration formula with a little modification.

Architecture for hardware

The function of computers in contemporary society is continually evolving, as was previously mentioned in the introduction and the chapter on the historical viewpoint. Nonetheless, the majority of the time, computers and other tech-based devices are made to benefit their users. This assistance was first primarily designed to aid with laborious activities, but gradually it evolved to improve user experience. The early computers were built to carry out activities that were technically or practically impossible to be completed by hand. For instance, the Electronic Numerical Integrator and Computer (ENIAC) was created to aid with complicated computations, which take a long time to complete and may include human error (see the part in Chapter 1 titled "The Early Computers"). Since then, computers have consistently been employed to provide a technical fix to a recognised issue. The earliest notable advancements in information systems were made in connection with industrial techniques. Without a computerised system, it is impossible to handle inventory (both incoming and exiting), warehouse management, production planning, and other tasks. Several MRP (material requirements planning) systems were created and commercialised during the manufacturing period in the 1960s. These systems support both production and planning by assisting with tasks like inventory control, bill of materials management, scheduling, and others throughout

the whole production process. Manufacturing what is needed at the precise time it is needed is made possible thanks to this planning. Planning well reduces organisational costs and boosts profitability. One of the factors that led so many industrial companies to adopt pricey computers was this increased profitability. Other more approaches for handling scientific problems were created concurrently. While their value may be assessed differently, these systems often contributed to the advancement of mankind in a variety of ways (weather prediction, safer cars and airplanes, the human genome, etc.). Later, as communications improved, the whole supply chain—including new marketing tactics and distribution methods—was discussed. A new layer has been added to modern information systems that addresses the consumer and the customised services offered to these clients.

The architectural evolution of computers was largely influenced by how information systems were thought to function. It was evident that some kind of storage device was necessary for the analysis of enormous volumes of data, as is the case with many systems. This sparked the creation of a wide range of solutions, all aimed at storing data so it may be accessed in the future.

The algorithms used to define the business rules must be converted into a collection of computer-understandable instructions. As a result, several programming languages were created, each having a set of fundamental capabilities as well as some unique characteristics. Several programming languages have been created throughout the years, but only a small number of them—including C, C++, C#, and Java—have gained widespread use. The processor (also known as the central processing unit), which mimics the human brain, is a crucial component in computer design. It is in charge of receiving orders, deciphering their intent, and carrying them out. The business rules must be converted by the developers into an ordered set of instructions that can be used by the programming language. In contrast to the human brain, which does most tasks intuitively unless they are part of a defined process like a computation, the processor always follows a structured programme. The processor's electrical circuits have the ability to interpret and carry out instructions.

Like the human short-term memory, a programme has to be in a working area in order to be run. In computers, the main memory is used to store both the data needed to run the programme and the programme itself. The computer requires some hardware that will act as the outer interfaces, much as with biological things. These gadgets serve two purposes. The different senses are employed as input devices, while other gadgets are used as output devices (muscles). In addition to these two kinds, computers feature hybrid devices, such communication lines or storage, that act as both input and output devices.

Most of the electronic gadgets in use today have been impacted by the quick technical advances of recent decades, particularly computers and other devices with built-in computers. When tracking the advancements of different mobile devices and the rate at which new, more compact, and efficient gadgets appear, this tendency becomes extremely evident. The phrase "computer generation" was used to describe the technical advancements that occurred in the early days of computing and how they influenced computers. As Moore's law anticipated, upgrading to a new generation entailed employing more dense electrical technology, which enabled faster processing and greater memory.

Generations of computers

Vacuum tubes were employed in the first generation (1946–1957) of computers, which had reliability concerns and were physically extremely huge, sluggish, and costly. A first-generation computer was the ENIAC (see the chapter 1 section on "The Early Computers"). While it was an improvement over the Harvard Mark I, it was still sluggish and produced a

lot of heat since it used vacuum tubes, which heated up like regular (inefficient) light bulbs. For instance, the ENIAC used 17,4681 of these tubes, which needed a significant amount of energy in addition to the heat output. The Electronic Discrete Variable Automatic Computer (EDVAC2) and the Universal Computer (UNIVAC I3) are two further computers from the initial generations.

Similar in idea, the vacuum tube was created concurrently with the electric bulb. But, the vacuum tube had two crucial characteristics that made it possible for computers to use them (and later in many other electronic devices). The tube may switch the signals and enhance them. The primary factor in the vacuum tubes' significant involvement in the development of early computers was their switching capabilities, which can be translated into turning on and off a specific bit. The heat emission was the biggest issue with employing the vacuum tube, however. Several times, despite attempts to cool the tubes, they overheated and ceased functioning. This had a significant negative impact on the system's dependability, together with the system's high number of vacuum tubes. The first-generation computers' capacity for software creation was likewise constrained. Just simple machine-programming languages were available, and even these were challenging to use. The systems had extremely few input and output device options and could only do one job at a time (see the section "Historical Perspective" in Chapter 1 for further information).

Second-generation (1958–1964) computers were distinguished by their use of the transistor, a vacuum tube replacement that had just been conceived at the time. It was created in 1947 at AT&T Labs, but it took some time before it was commercialised. The transistor was more dependable, smaller, quicker, and, most importantly, substantially cheaper. The transistor is built of silicon, which is readily accessible and reasonably priced since it can be extracted from sand. The single transistor creates nearly little heat, in contrast to vacuum tubes, which increased the system's dependability and reduced the cost of cooling it. The development of the transistor had a significant impact on the whole electronic sector as well as computers. The advent of the transistor and the subsequent downsizing techniques made it feasible for even the fast advances in space technology throughout the 1960s. The usage of single transistors, as it was with the second generation of computers, lasted just a few years, despite the fact that transistors significantly contributed to hardware implementations.

The use of symbolic programming languages on second-generation computers sped up the development process. Other research and development efforts were concurrently directed towards special-purpose programming languages like FORTRAN, which was designed for scientific contexts, and COBOL, or Common Business Oriented Language.

Integrated circuits were a feature of third generation (1965–1971) computers. The integrated circuit allowed for extra substantial progress despite the fact that the transistor represented a quantum leap in hardware development. The integrated circuit, often known as a semiconductor chip, was created concurrently by two unrelated inventors: Jack Kilby, a Texas Instruments employee who won the Nobel Prize for his discovery, and Robert Noyce, a Fairchild Company employee. A group of transistors that have been integrated and crammed onto a single chip called an integrated circuit. Such a chip dramatically boosts the system's performance while cutting expenses. It now has hundreds of millions of transistors. Since its creation, integrated circuits have advanced, and every 18 months or so, the number of transistors doubles while the price stays essentially constant (see Table 1.2 and the Moore's Law section in Chapter 1 for more information).

Third-generation computers came with the "usual" input and output hardware, including keyboards and mouse, as well as more advanced operating systems that could handle many tasks at once. The main advantage was that adopting integrated circuits allowed for cheaper

prices, which made computers more affordable for a larger user base.

The development of the microprocessor and a large rise in the number of transistors packed into a single chip were the defining features of fourth-generation (beginning in 1972) computers. It uses a single chip to construct a fully functioning CPU. The performance and speed improved by packing hundreds of millions of transistors onto a single chip. The time required for electrons to go through the processor is impacted by the shorter distance, which further reduces the duration. Ted Hoff, who worked for Intel (a company founded by Noyce, the inventor of the integrated circuit among other things), is credited with creating the microprocessor, which at its inception was no bigger than a pencil sharpener. Ted Hoff is also responsible for the widespread distribution of computers as we know them and the fact that almost any human activity is done, managed, monitored, and even billed by some computer. The new processor was created with the intention of being used in the creation of calculators. In the end, however, it turned out to be far more significant since it helped bring about the widespread use of computers that we are all familiar with. The word "generation" was superseded by measuring the transistor gate length in nanometers for the fourth generation of computers, which are now in use.

CHAPTER 8

CLASSIFICATION OF COMPUTER

Ramesh S, Professor,
Department of Computer Science and Engineering,
Jain (Deemed to be University) Bangalore, Karnataka, India
Email Id- ramesh.s@jainuniversity.ac.in

A clearer description of the many kinds of computers or a better categorization was necessary due to the computing industry's fast technical progress. Yet in the beginning, there were simply "computers. The term "mainframe" was used to describe and categorise the corporate computers in charge of back-end processing. As a result, these systems had to provide a high level of dependability, availability, and serviceability since the performance of this computer was essential to all organisational functions. Only when alternative computer architectures, such the departmental system, began to take shape did the word mainframe come into use. Initially, a single department was meant to be supported by the minicomputer or departmental system. The minicomputer was developed by Digital Equipment Corporation* with the intention of serving organisational divisions that were given less attention by costly organisational structures. This market segment, which began as a marketing concept, turned out to be quite lucrative. Departments could afford to purchase their own system, which was substantially less expensive and offered excellent service to individual departments, rather than spending enormous sums on an organisational system that gave poor service to the departments.

The microprocessor chip served as the foundation for the personal computer, which was originally designed for a single user. Microprocessor chip-based computers are now utilised by many systems, including huge ones like mainframes (which in certain instances have been superseded by a new term—servers), and are not only used by a single user.

It is uncommon to apply the usage-based categorisation indicated above since it is inaccurate. Additionally, the taxonomy needs to be expanded to cover more categories because of how widely we use computers in our everyday lives: Supercomputers are referred to as incredibly fast and powerful computers that are capable of conducting complicated large-scale applications like molecular modelling, numerous simulations, and weather forecasting. In the past, the design and creation of systems with specialised functions, such vector processors, was the preeminent technique for meeting these objectives. † Today, parallel systems built on an array of cooperating microprocessors are used to meet all of these needs. Another example of high-performance systems in use today is cloud computing, which gives users access to a virtual system made up of a variable number of processors and located someplace on the network. Servers, which in many businesses act as a partial substitute for mainframes, primarily for a particular purpose, such as a mail server, file server, print server, and so on. These servers may be thought of as the "common denominator" between the departmental systems and the mainframe. On the one hand, some of the server's burden, which was more appropriate for the task, replaced some of the mainframe's workload (see Chapter 1, "Introduction and Historic Context"). On the other hand, servers took the place of the departmental systems. Because of this, there is no longer a distinct boundary between mainframes and departmental systems.

Typically, midrange computers are bigger servers that serve a number of departments or even certain internal users of the company. They were often more compact than mainframes but offered more features than regular servers.

Workstations are several types of personal computers used for a certain organisational job. The particular setup and devices connected to the system were determined by the workstation's purpose. For instance, CAD (computer-aided design) workstations needed to have a faster CPU and more memory to accommodate running engineering models. Moreover, the system would have better graphics capabilities and often needed a larger screen with greater resolution. At the moment, every workstation is a personal computer or even a tablet.

The majority of electrical and electronic equipment, such as washing machines, television sets, alarm systems, watches, cameras, and many more, are considered appliances and are often controlled by computers. The cellular phone, which has taken over as the primary network access device, entertainment device, and mobile information centre, requires special attention owing to its widespread use.

Systems for Computers

Each computer system comprises of the following parts:

Processor, accountable for carrying out the commands (the brain in the human body) Memory, which serves as a temporary repository for the data and instructions needed to carry out activities (the various memories in the human body) channels connecting the different components for conveying data (the circulation and nervous systems in the human body) input and output devices that link the computer to users or the outside world (in the human body, corresponding to the senses, which serve as input devices; and the muscles, which serve as output devices) Every computer has these components, and the particular kind may have an impact on their amount or other characteristics. For instance, a big server that is designed to accommodate several concurrent active users would likely have more memory.

A computer system with the aforementioned components is schematicall. The ycentral processing unit (CPU) and two units are on the left side (the arithmetic and logic unit [ALU] and the control unit [CU], as well as other components to be explained later). The memory and two boxes that represent the input and output devices are additional system components. The channels over which the data is conveyed are represented by the lines that link each of these parts.

Embedded device: Even the computers used in embedded systems are of the same kind of component, however their characteristics may vary owing to the various use. Such a setup, and as can be seen, the input and output elements may need actuators and sensors (depending on the type of system). For instance, an alarm system may need a lot of connections to different sensors and relatively little memory (motion, volume, smoke, etc.) The von Neumann architecture is still used by the majority of contemporary computers (see "Von Neumann Architecture" in Chapter 1 for further information). Architecture design made strides when the concept of processor and memory separation was introduced. This division gave rise to the notion of loading a programme, or the idea that the computer may be utilised for a variety of activities. This emerged from this isolation and is one of the distinctive qualities of computers. While the hardware is unaltered, the computer's behaviour may be altered by loading and running various applications.

The architecture that was suggested included registers, which are tiny, quick pieces of memory that are housed inside the processor and are used to store data needed during processing. Registers can, for example, hold the address of the next instruction to be

executed, the next instructions next to be executed, or they can aid in arithmetic operations.

An extra separation between two kinds of memory was given considerable consideration in the Harvard design. The memory used for instructions is one kind, whereas the memory used for data is another. Additional channels that could transmit data and instructions concurrently were needed because of this isolation. Better and more dependable performance is offered by this extra degree of parallelism. The operand required for the executions might be brought by the processor while it is fetching the following instruction. The simultaneous operation of these two transfers across two separate channels is possible. Further possibilities are provided by separating the memories and the fact that each one has a unique channel. These might be heterogeneous channels with various features that could be helpful in creating architectures more suited to certain requirements. Because of its greater performance predictability, the Harvard design serves as the foundation for many signal processing systems. This is made feasible by the channel having fewer potential collisions. This architecture is also employed in many real-time systems because to its increased predictability. Real-time systems, in contrast to "regular" systems, must ensure execution inside a specific temporal window.

Processor

The system's brain, which carries out the commands, is the processor. The instructions are divided into parts, or microinstructions, to make the execution process simpler. It is possible to divide an operation into several phases during execution, which may boost speed via parallelism (this will be covered in more detail in Chapter 4, "Central Processor Unit"). We will assume that each instruction is composed of four microinstructions in order to better comprehend the notion of microinstructions and to make the explanation simpler: Fetch is the step in which the CU transfers the instruction from memory to the processor. The instruction must first be fetched since the von Neumann architecture divides the CPU and memory.

Decode refers to the process by which the CU decodes the instruction to ascertain its validity, the kind of instruction it is, the number of operands it needs, and their locations. The ALU cannot retrieve the instructions or bring the necessary operands; it can only execute the instructions. The CU is required to do all the preparation work as a result. Only after the CU has brought the necessary operands and placed them in additional special-purpose registers in the CPU and stored the instruction, before signalling the ALU that it may begin the execution, does it do so. As not all instructions have the same length or need the same amount of operands, the decode step is essential. For instance, there is just one operand in the JUMP instruction (the address). Although certain instructions, like NOT, only need one operand, arithmetic instructions typically require two operands and sometimes three.

The ALU performs the instruction utilising the operands that were delivered by the CU during the execute step. Write back refers to the process of returning the instruction's result to memory or the target register. The CU is required for this transfer since the ALU cannot access its exterior environment (memory or devices). The initiation phase, which consists of the first two stages since the CU performs them, is one of the four phases and serves to get the instructions ready for execution. As it comprises the executions (carried out by the ALU) and the storing of the result, the second phase—which is made up of the final two stages—is called the execution (performed by the CU). A programme is really run by continuously carrying out each of its instructions one at a time. The CPU never sleeps, and even when it has nothing to do, it performs an infinite loop of idle instructions that don't really do anything but are there for the sake of the cycle. Because of this, following instructions is a never-ending procedure (as long as the system is up).

Center for Processing

The many facets of the processor (sometimes referred to as the central processing unit [CPU]). The processor, which can carry out the duties specified by the programmes, serves as the brain of the system. The precise placement of the CPU may be determined by comparing it to the generic hardware architecture diagram. The processor is able to "understand" and carry out the program's instructions thanks to its electrical circuitry. Each processor is broken down into additional components due to the use of a modular approach, as previously mentioned (see the section in Chapter 1 on the Von Neumann architecture and the processor). These additional components include the control unit (CU), the arithmetic and logic unit (ALU), a special and dedicated internal fast memory, and others.

By bringing each instruction from memory, decoding it to determine its operands, fetching these necessary operands, and then passing it to the ALU for execution, the CU is in charge of scheduling the instructions to be performed. The operands of the ALU are used to carry out the instruction. Inside the processor, registers are one type of quick temporary memory. There are many register types; some are accessible to programmers, others are only accessible to operating systems, and a third kind is exclusively utilised by the hardware itself (for further information, see "Attributes of the Early Computers" in Chapter 1). It should be emphasised that accessing registers is only possible with the use of assembly language as part of software abstraction. The majority of high-level programming languages do not provide this feature; instead, the compiler is the sole tool that can use the registers.

Registers

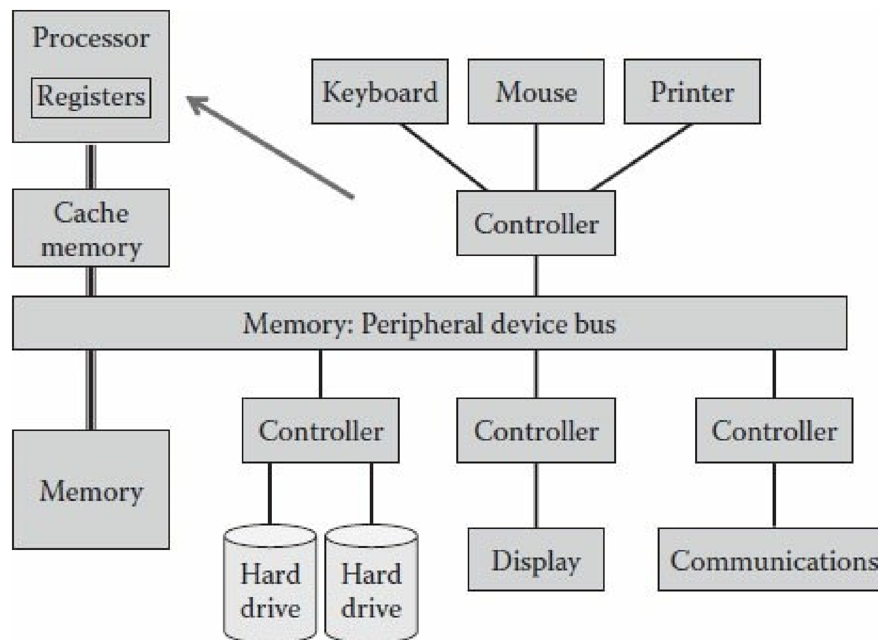

Figure 8.1: The CPU as part of the system.

The desire to make computers run more quickly gave rise to the concept of registers. Since the design flaw of the CPU being far quicker than the memory existed even in the early days of computers, it had to be fixed. The CU must retrieve each instruction and its operands from memory in order to execute it. Due to the relatively sluggish memory, even the fastest processors will need to wait for the operands to be retrieved. Since operands are used in the majority of operations, memory access becomes a key speed-limiting issue. The

implementation of registers was the first and most successful way to solve this problem over the years. Other solutions included read ahead, in which the central unit reads the instruction that will be needed in the near future (this will be explained in more detail later in this chapter), and cache memory (Figure 1).

A register is a tiny, very quick memory that is located within the CPU. It accelerates the execution of the programme by offering a very quick access time to the data contained in the register. The registers mimic the short-term memory of a person in certain ways. Registers have a quick access time, a small amount of storage, and are solely utilised for transient data, much as short-term memory. Registers are regarded as the top of the memory hierarchy because of their speed (this term will be explained and elaborated in the next two chapters). Initially designed to enable hardware operations, registers are now used often by contemporary processors as part of daily activities. For instance, every system should have an internal register that stores the location of the next command to be executed (also known as a programme counter, or PC). The address of the subsequent instruction is found by consulting the PC when the CU needs to retrieve it for execution. The queue number display that is often employed by different service firms may be seen as the PC. While its content is changing in accordance with software activity, this register is maintained by the hardware and cannot be altered by software (applications or operating systems). Every loop, for instance, has a conditional branch and has the ability to alter the contents of the PC register in some manner. Additionally, the ALU solely uses internal registers while carrying out the commands. This implies that the operands of the instruction are really stored in these internal registers by the CU that fetches them. The result is stored in another internal register when the ALU has completed the execution, and once again, the CU is in charge of transferring its information to the necessary location (as defined in the instruction).

Assuming, for instance, that the command to be carried out is: During compilation, the instruction will be converted into a binary mnemonic that represents the ADD instruction, regardless of the particular programming language being used. In this instance, there are three variables A, B, and C and the command is intended to replace the value of variable C with the sum of variables A and B. The binary mnemonic will include all pertinent data required for execution, including the ADD instruction code, addresses for the first and second operands (A and B), and addresses for the result (C). The CU will get the instruction, translate it into an ADD instruction, and recognise it. Since ADD has three operands, or parameters, it will retrieve the first two and store them in the internal registers of the ALU. It won't notify the ALU to do the add operation until that point. The CU will duplicate the outcome that was saved in the internal output register of the ALU to the third parameter specified in the instruction when the ALU completes (the variable C).

There are registers that are intended for usage by the programme, as was previously described. Registers are a common component of computer design and are present in high quantities in contemporary computers due to their significance in improving performance. Register use did not suddenly become popular; there were earlier technical alternatives. The original computers were built on the concept of a stack and operated without any registers. The instructions presupposed that both the result and the input operands were placed on the stack. The hardware instructions had to be modified in order to support and access this new sort of temporary memory when the new computers that featured registers began to appear. The first implementation only made use of one register (sometimes referred to as the accumulator due to its usage in arithmetic instructions). Later, with the advent of general registers and registers for specific purposes, including data registers and address registers, the utilisation expanded.

Architecture Based on Stacks

In comparison to modern computers, the early computers were very rudimentary (even primitive), and they were built using a stack design. This indicates that the instruction consisted just of the operation's mnemonic without any operands. This explains why this design is sometimes referred to as the architecture of no-operand instructions. The operand (one or more) was often located at the top of the stack according to convention. This implied that before starting the operation, the developer had to add the data operands to the stack. The CU retrieves the instruction in this instance, decodes it, and then determines the necessary number of operands.

The CU will utilise the two operands at the top of the stack in the preceding example if the instruction is ADD and there are two operands. There are additional circumstances in which just one operand is necessary, such as an instruction like

As can be seen, just the outcome is saved on the stack; the original operands are no longer there. There are two explanations for this conduct: By utilising the conventional POP circuitry, the CU may be made simpler and the stack won't be clogged with extraneous data if the operands are removed.

The last instruction in this sequence transfers the result into the variable C after removing it from the stack. The stack is once again empty, and the arrow depicts a data migration from the stack to memory.

The stack-based design was created to accommodate the Polish notation, which was created by Polish philosopher Jan Lukasiewicz at the beginning of the 20th century and is also referred to as prefix notation. Polish notation places the operators to the left of the operands when defining logic or mathematical formulae. Instead of the usual formulae, where the operators are sandwiched between the operands, this is used.

For instance, the traditional notation for adding two integers is expressed as:

$$\mathbf{A + B}$$

By using Polish notation for the same formula, we will have to write:

$$\mathbf{+A\ B}$$

In the early days of computers, the Polish notation was quite popular and even employed by a family of programming languages (Lisp, for instance); some of them are still in use today. Recursion, trees (a kind of data structure), dynamic typing, and other key concepts in computer science were among the first to be defined in the Lisp programming language. The stack-based design was created to provide the Polish notation a straightforward infrastructure.

A mathematical formula that adds four numbers such as

$$\mathbf{R = A + B + C + D}$$

when using Polish notation will change to

$$\mathbf{R = + + +D\ C\ B\ A}$$

In addition, the stack-based architecture instruction will be:

PUSH	D	# Push variable D to TOS
PUSH	C	# Push variable C to TOS
PUSH	B	# Push variable B to TOS
PUSH	A	# Push variable A to TOS
ADD		# Add the two items on TOS (A + B); store result on TOS
ADD		# Add the two items on TOS (A + B + C); store result on # TOS
ADD		# Add the two items on TOS (A + B + C + D); store result on # TOS
POP	R	# Move TOS (A + B + C + D) into variable R

It can be seen that it is a straightforward translation. Although the stack-based architecture was replaced by other, more efficient architectures, it influenced the computer science discipline (e.g., by supporting software-based stacks and instructions to handle them).

Architecture Based on Accumulators

The next phase of CPU development aimed to get around these limitations and boost execution speed. The gap between the speed of the processors and the speed of the memory widened as processor speed grew. Certain performance restrictions imposed by the stack, which was a component of memory, had to be removed. The development of a data register known as the accumulator provided the answer. The access time was greatly reduced since the accumulator was a component of the CPU. In accumulator-based architecture, the accumulator was used for all arithmetic operations, and the result was also stored there. The outcome might then be ready to be used as input for the next instruction. In contrast to the stack-based architecture's arithmetic instructions, which lack any operands, the accumulator architecture's arithmetic instructions do contain one operand. It is expected that the second operand is present in the accumulator. This suggests that at least one operand must be present in the accumulator before an arithmetic instruction may be carried out. It is the developer's obligation to load the operand into the register while developing an assembly programme. The compiler handles it for higher-level programming languages. The prior content (one of the operands), which was stored in the accumulator, is deleted since the result is placed there. Moreover, the instruction's result must be copied from the accumulator to avoid being replaced by the operand or result of the next instruction. Its behaviour resembles that of short-term memory. If the information in the short-term memory is significant, it must be moved to the long-term memory; otherwise, it will be lost and replaced by fresh information.

Architecture Register-Register

The expansion of registers and the speed benefits they provide prompted yet another architectural improvement. The accumulator and eventually the extra registers were designed to provide a quick temporary memory inside the CPU. The performance improvements of the earlier designs (accumulator-based architecture and memoryregister architecture) were constrained since one operand resided in memory. The next stage in systems architectural design was to stop using memory for operands and start using solely registers. The operands

of the arithmetic instructions are stored in registers in the register-register architecture, reducing the requirement to enter memory. An instruction of this kind typically has three operands (registers). The input registers make up two of the operands, while the output register makes up the third. The drawback of the register-memory architecture, which ignores the first operand, was overcome by this structure of the instructions. The register-register design enables choosing a new destination register and does not keep the result in the input register while overriding the prior value.

Processing Routes

The processor consists of a number of related parts. The necessary data is sent before, during, and after the instructions are executed via the pathways that link all of these components. These roads mimic a system of roadways that link locations, except instead of moving automobiles, the computerised paths move data. A simplified illustration of the data exchange needed for a straightforward ADD operation is shown in Figure 8.2. The register-register architecture, which includes this ADD instruction, was discussed in the section before this one. Several of the general-purpose registers are shown in the figure 1 top portion. The figure 1 only refers to four of these registers, despite the fact that the CPU may contain many more. The operands must be positioned in the input registers before the instruction is sent.

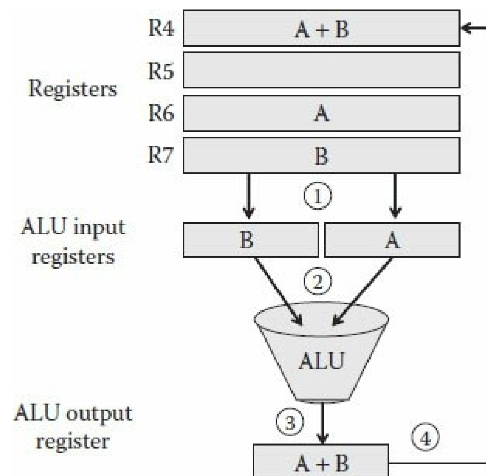

Figure 8.2: Processor's paths.

The two operands in this example, A and B, are in the correct registers (R6, R7). In this instance, the guidance was the CU gets the instruction as part of getting ready for execution, decodes it, and discovers that R6 and R7 are being used as operands. The two operands are transferred to internal registers inside the ALU before the ADD is started, assuming the ALU has completed processing the preceding instruction (Figure 8.2). The output of the ALU, which is shown as a trapezoid cone, is first stored in an internal register before being transferred to the target register (R4).

CHAPTER 9

VON NEUMANN ARCHITECTURE

Rajesh A, Professor,
Department of Computer Science and Engineering,
Jain (Deemed to be University) Bangalore, Karnataka, India
Email Id- a.rajesh@jainuniversity.ac.in

The Von Neumann architecture, as was previously said, separated the different components of computers, and contemporary computers maintain this tendency of isolation. The ALU is further broken down into specific parts (as will be further elaborated in this chapter). Such components each have own internal registers. Because of this, the general-purpose registers, which are also a component of the processor, are segregated from the ALU, and the necessary copy operations are handled by the CU. There are two different register types. The general-purpose registers are in the top section, while the input and output internal ALU registers are in the bottom section.

The picture illustrates many paths is designed to move the operands of the instruction from the general-purpose registers to the internal registers of the ALU (marked as 1 in the diagram). Only the CU has access to these pathways as part of the execution preparation. Paths designed to transport the information from the internal registers into the ALU (marked as 2 in the diagram). The transfers are carried out as part of the execution of the instruction, and these routes are only available to the ALU. a way to move the output (the outcome) from the ALU into the internal register of the ALU (marked as 3 in the diagram). The transfer in this route is from the ALU into the internal register as opposed to the transfer in the path that was previously described, which was from the internal registers into the ALU. Similar to the preceding instance, only the ALU itself may access this route.

A way to move the output from the ALU's internal output register to the destination register for general use (marked as 4 in the diagram). This form of route is similar to the first type of pathways stated before, however in this instance the path is for the output rather than the input operands as in the first type. Only the CU has access to this route.

Implementation of instructions: The instructions the processor can carry out establish the capabilities of the computer system. Because of this, the instructions had to be modified as computers advanced in order to provide software programmers the extra capabilities they needed. As was previously mentioned, the first computers used extremely basic instructions, such as stack-based architecture, where the commands had no operands. But, as technology improved and new applications were created, so did the needs. These applications were initially rather straightforward, but as time went on, they became increasingly intricate. Need for increasingly complex applications was what sparked end-user demands for increased capability. Developers looked for more effective and resourceful software solutions to aid in the development process in order to fulfil the dwindling deadlines set by the shifting market. The result was a new generation of hardware-supporting instructions that were more reliable and had improved functionality. In order to address this, hardware developers often added new features. As it takes time and effort to create, develop, and implement new instructions, the majority of manufacturers have embraced the von Neumann architecture's modular concepts. During the execution of the single instruction, which was broken up into a number of preset phases (or building blocks), one component of modularity was implemented:

Calculating the memory address and bringing the relevant bytes comprising the instruction and its operands together is the process of fetching the instruction. Identifying the number of operands an instruction contains after decoding it in order to determine if it is a valid instruction. Copying the operands into the internal ALU registers from their destinations (memory and/or general-purpose registers). Directing the ALU that all input data is accessible before issuing the command to be carried out. Transfer of the outcome from the ALU internal register to the specified location (memory or register).

Modern processors split the execution into many more stages, allowing for a greater degree of parallel processing, in order to increase the execution speeds (this will be discussed later in this chapter). The division of the training into these predetermined phases offers an extra crucial capacity, nevertheless. Similar to the Lego construction games for kids, each level may be seen as a set of building blocks. It is easier to add new instruction when using these preexisting building pieces, also known as microinstructions. In fact, this solution gives hardware engineers powers like those of software developers. Similar to how the hardware engineer utilises these microinstructions to create new instructions, the software engineer uses the hardware instructions as the foundation for the programme that will be created. This idea is presumably related to software development principles, which call for functions or methods to be highly compact and execute only a single purpose. Software experts will probably find this to be the case. This is one of the more well-known methods for handling complexity, both during the initial stages of development and afterwards in the maintenance phase.

Performance: One must quickly discuss the topic of performance in order to comprehend the trends in processor development as well as the microinstructions technology. The need for new and complex applications is continuously growing, and system developers are constantly seeking for hardware with a greater performance level. Performance's definition, however, is not precise. As with many other technologies, there are several definitions of a technology's performance when there are numerous implementations of the technology. For instance, one sort of performance described when talking about autos is the maximum weight the vehicle can support. Driving speed is an additional performance factor that is often relevant for smaller family automobiles. For vehicles with specialised uses, such as racing or sports automobiles, acceleration speed may also be relevant. Of course, there might be more criteria used to gauge the performance of the autos. The similar issue arises when attempting to provide a universal definition of computer performance.

As each kind of user may have distinct demands, we must determine what is crucial for each in order to more accurately analyse the performance problem. For instance, a user using a local computer that is linked to a distant server wants to have the fastest response time possible. † Performance may be defined differently by the management of a compute centre. The total system throughput is crucial in addition to the reaction time, however. The fundamental reason is that in order to get a better price/performance ratio and to cover the significant expenditures connected with the centre, a big number of users are required. The problem is significantly more complicated when considering web-based apps and the technology that supports them. Even for internal usage, a slow reaction time in such a system would increase the amount of time spent waiting. Given that human resources often account for a significant portion of the budget in service-oriented firms, a slow-responding system results in poorer productivity and the waste of costly resources. Much more challenging are web systems designed for users. As the services or goods of the rivals are just a few clicks away, the issue here goes beyond the systems' reaction times and encompasses the whole browsing experience. There is no universally accepted definition of performance when it comes to computer systems, much as in the example of automotive performance, because of

the variety of functionality offered by the system. The many contemporary portable gadgets (tablets, smartphones, etc.) that redefined performance to include user experience are a good example of this. This development transformed the way computers function and was effectively advertised by Apple as a key differentiation and selling point. Yet, a mechanism for evaluating various systems was essential throughout the evolution of computing technology, particularly given the expensive nature of computers. The methodologies took the form of a number of measures that offered a common framework for comparison. This led to the following definitions:

The amount of instructions a processor is able to execute in a second is measured in millions of instructions per second (MIPS). Even if it is inaccurate, it offers a general comparison of several systems. It should be emphasised that instructions in this context do not refer to high-level programming language instructions, but rather to machine instructions. The compiler converts the high-level language instructions into machine language instructions. One high-level instruction is often broken down into numerous machine-level instructions. MIPS measurements may be deceptive from a software or systems engineering perspective since it is uncertain if the mix of instructions used in the test run accurately represents the instructions utilised by the application.

A measurement based only on floating-point operations is called millions of floating-point operations per second (MFLOPS). This measure was created to distinguish between systems for general use, such as the majority of management information systems, and systems used primarily for research reasons that need this sort of instructions (MIS).

Megabytes per second (MB/sec): This evaluates the capacity for data transport, including the input and output devices or internal pathways (buses). Of fact, this is a unique performance statistic that only indirectly affects how quickly the programme runs. The primary measure of the metric is the volume of data that can be processed in a second. The execution will be slower the longer the CU waits for the instruction or operands to transfer from memory owing to a slower bus, as will be covered in the bus chapter. Because it doesn't matter what kind of data is carried, this statistic is correct from the perspective of systems engineering. The pace won't change. The performance of the system as a whole does not, however, directly correlate with the data transmission speeds.

Transactions per second (TPS): This gauges how many transactions the system can process in a second. This statistic was created along with the creation of several concurrent user web applications, such as those for reservations, shopping, and banking. It is necessary for the transactions to be precisely characterised for an accurate TPS measurement (content and execution frequency). Predictions based on the TPS metric are more trustworthy in terms of systems engineering since it takes a more holistic approach than the other three metrics. The sole factor to take into account is the kind of transactions made, since query transactions are easier and quicker than database update transactions. Even with the tiny sample size indicated above, it is possible to comprehend the variety and the wide range of special requirements despite the many additional measures that have been created over the years. It should be emphasised that the measures were created by different vendors in an effort to maximise the likelihood that their systems would be purchased. The measures were designed to aid in decision-making since not every client had the skills and knowledge necessary to assess the performance of the different systems. A vendor would have a greater chance of selling a system to potential clients if, for instance, he or she could prove that it had a higher TPS value.

Inside-of-the-processor clock: An internal clock keeps the activity of the CPU in sync. The monitoring capabilities necessary for carrying out the instructions and coordinating all the

components are provided by this clock. For instance, before the ALU begins the execution, the CU must be in sync with it and provide all input. The cycle time of the system is determined by the clock frequency (or clock rate). All of the processor's operations are synchronised using these cycles. As the meaning of the clock frequency and the cycle time are identical, one number may be calculated from the other,

$$\textbf{Cycles per Second} = \frac{1}{\textbf{Clock time}}$$

For instance, if a system's cycle time is one millionth of a second, it follows that the clock in that system "beeps" one million times per second. The frequency of the clock, or second cycles, is indicated by these "beeps." The clock cycle is the interval between two beeps (Figure 1). The execution time of a particular application may be measured with accuracy using the clock frequency. This is accomplished by keeping track of the execution's beep count. The time needed for the programme to execute may be calculated using the preceding example by multiplying the number of beeps by one million (the number of beeps per second).

Visual representations of the cycle time and the clock time are shown in Figure 1. Early computers needed many cycles to complete an instruction. Nonetheless, several instructions could be carried out in a single cycle with current computers. If we take into account a CPU that operates at 1 GHz, this implies that the cycle time is 1 billionth of a second (109) and the processor "beeps" one billion (109) times per second. It is feasible to utilise the number of beeps for comparing the execution of two apps since the number of beeps and time may be interchanged. Nevertheless, it should be emphasised that only programmes running on the same system are eligible for this assessment. The amount of beeps is inadequate for a precise assessment if the programmes are running on separate platforms. It is feasible for each system to have a distinct clock frequency when the applications are run on various systems. In this situation, focusing just on the cycles (or beeps) is incorrect. We must multiply the number of beeps by the cycle time in addition to the number of cycles when evaluating execution timings (Figure 9.1).

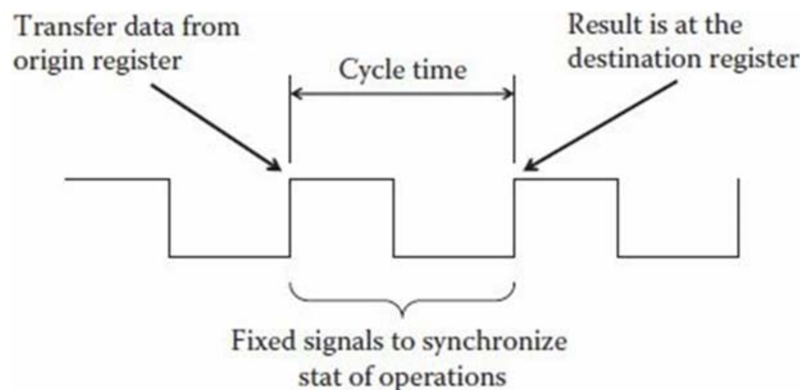

Figure 9.1: Shows the Clock time.

As the clock frequency is a hardware characteristic, it remains constant during the execution of many programmes. In light of the fact that all of these scenarios include the same clock frequency, the number of cycles (or beeps) is enough for measuring or comparing the execution periods of various programmes that were running on the same system. This concept is straightforward to mathematically demonstrate. The amount of seconds needed to execute the programme defines the processor time. This time is estimated by multiplying the amount

of cycles needed to execute the programme by the cycle time,

$$\textbf{Processor time} = \textbf{Seconds per application}\left(\frac{\text{Seconds}}{\text{Application}}\right)$$

$$\frac{\text{Seconds}}{\text{Application}} = \frac{\text{Cycles}}{\text{Application}} * \frac{\text{Seconds}}{\text{Cycle}}$$

“Iron Law” of Processor Performance

While the word "performance" might imply different things to different users, the most prevalent definition refers to the reaction time the system achieves. Nevertheless, this time also includes any other time-consuming tasks that could be necessary, such input, output, and communication, in addition to the time needed to complete the specified job (run the programme). In this part, we will just focus on the processor performance, or how long it takes the CPU to complete a given job (or program). We shall specify it in this instance

Processor performance = time per program

However, the time per task (or program) is calculated based on three parameters:

The program size or the number of instructions executed in this specific run of the program

Cycles per instruction (CPI): The average number of cycles required for executing one instruction Cycle time: The amount of time for each processor cycle As such, the formula (“Iron Law”) to calculate the processors performance is:

$$\frac{\text{Time}}{\text{Program}} = \left(\frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}\right)$$

$$\textbf{Time} = (\textbf{Program size} * \textbf{CPI} * \textbf{Cycle time})$$

The size of the programme, or the number of instructions to be executed, is determined by the first variable in the formula. This value is determined by the developers' software and cannot be altered by the hardware. In this sense, the hardware takes it for granted that there will always be a fixed number of instructions to be processed. In course, the quantity of instructions that are executed may vary depending on the input. The programme will need to be examined, and the time-consuming section will need to be rebuilt using a more effective approach, if it is necessary to reduce this number, for example because of lengthy execution times. As an alternative, adopting a more complex compiler can result in a code that executes fewer instructions overall. Sometimes the time required to translate high-level programming language instructions into machine-level instructions may be sped up by the compiler, for instance by removing superfluous code or making better use of registers. An important enhancement component that has evolved through time to speed up execution is the second variable in the formula (CPI ratio). The performance of the processor is directly impacted by lowering the number of cycles needed for a single instruction. The typical CPI in the 1980s was 5, meaning that five cycles were needed to complete a single machine instruction. On the other hand, modern processors have a CPI of less than 1, which indicates the processor is able to execute many instructions concurrently during the same cycle.

Another significant criterion for improvement that many computer makers take into consideration is the third number (cycle time). The clock rate has increased by more than three orders of magnitude during the last three decades. Yet during the last ten years, the tendency towards shorter cycle times gave way to the trend towards more processors or cores.

The many execution units provide significantly higher performance improvements when combined with the trends in software engineering that use threads*. With this succinct description of performance, we may go on to a more comprehensive examination.

It signifies that the performance of X is n times that of Y if processor X is claimed to be n times quicker than processor Y. The execution time on processor Y, however, will be n times longer than the execution time on processor X since performance and execution times are inversely related. However, this straightforward extrapolation has certain drawbacks. Would it be accurate to say that processor X is twice as fast as processor Y if a given application runs on processor X for 10 seconds and on processor Y for 20 seconds?

Several of the kids will undoubtedly respond "yes" without any hesitation, however the correct response is no! The performance of the CPU cannot be estimated from the data gathered from executing a single application. For instance, if this test programme doesn't make use of any floating-point instructions, it won't be able to tell how well the processor performs in floating-point operations. The right response is that processor X was twice as quick while executing this particular test application, but we are unable to evaluate its performance with other programmes. There are many different approaches for evaluating a processor's performance since there isn't a single accepted benchmark or set of criteria. These techniques, developed by diverse manufacturers, gave them the ability to better showcase their systems. It should be emphasised, however, that the majority of the systems in use today are based on single-user personal computers. As these systems are reasonably priced, the evaluation criteria are not really necessary. Decisions back then were considerably more important in terms of the performance to be obtained and the cost to be paid since there were many suppliers offering a choice of highly pricey systems. Even if the older measures and approaches are seldom employed, they are still accessible in many other situations.

Instruction-Based Metrics for Cycles Per

The number of cycles needed to complete each instruction is determined by the processor architecture. A lesser number of cycles will be needed for simple instructions compared to more cycles needed for complicated instructions like division. The CPI may be used as a more precise statistic to gauge the performance of the projected system since it is continuous (per each instruction). We have to calculate the mix of instructions utilised in a particular programme when utilising a CPI-based measure. The various instruction kinds and their occurrences will be included in this mixture. It is feasible to determine the average CPI by utilising such a mixture. The average CPI is a dynamic that is directly generated from the mix of instructions, in contrast to the CPI, which is a constant value for each instruction (the number of cycles needed to execute it). As a result, the average CPI offers a more precise evaluation for the particular run. The "Iron Law" may still be used to determine the overall

$$\textbf{Time} = \textbf{Average CPI} * \textbf{Number of Instructions} * \textbf{Cycle Time}$$

Execution Time

The advantage of using this metric is that it is accurate for the specific run and reflects the actual usage of the system. On the other hand, it is quite complicated to estimate the instructions' mix and if the estimate is not accurate it will hamper the results obtained.

Example:

Assuming a program was run on two processors (A and B), using the same compiler.

The cycle time of processor A is 1 ns (10−9s). While running the program the average CPI obtained was 2.4.

The cycle time of processor B is 1.5 ns and the CPI measured was 2.

Which processor is faster? And by how much?

In solving the question, we'll have to use the "Iron Law":

$$\frac{\text{Time}}{\text{Program}} = \left(\frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}\right)$$

$$\textbf{Time} = (\textbf{Program size} * \textbf{CPI} * \textbf{Cycle time})$$

In this specific case, the number of instructions that were executed is unknown, but since it is the same program and the same compiler, it is safe to assume that the number of instructions executed was roughly the same. We will mark the number of instructions by N and then,

$$\text{Time}_A = N * 24 * 1 = 2.4N$$

$$\text{Time}_B = N * 2.0 * 1.5 = 3.0N$$

$$\frac{\text{Time}_B}{\text{Time}_A} = \frac{3.0N}{2.4N} = 1.25$$

This indicates that the amount of time needed to execute the programme on processor B is 25% longer than the amount of time needed to run the programme on processor A. Processor A is thus 25% quicker than Processor B for this particular software. Systems engineers may evaluate the performance of various processors using the prior example. Nevertheless, the CPI measure may also be used to determine what hardware modifications will be necessary to make one CPU perform similarly to another processor, in addition to comparing the relative performance of other processors. These necessary adjustments, which mostly affect hardware engineers, are included here for completeness.

Consider the previous example: what should be the CPI on processor A so the performance of the two processors will be identical Once again, we will use the "Iron Law" and figure out the new CPIA:

$$\frac{\text{Time}_B}{\text{Time}_A} = 1$$

$$\frac{\text{Time}_B}{\text{Time}_A} = \frac{(N * \text{CPI}_A * 1)}{(N * 2.0 * 1.5)}$$

$$\text{CPI}_A = 3.0$$

As a result, if processor A's average CPI is 3.0, the two systems' performance will be equal. It should be mentioned that there are further alternative adjustments that might be made for the same performance. Both the CPIB and the cycle times of the two processors are adjustable. Thus, we will utilise the formula above but alter the unknown if we wish to adjust CPIB instead of CPIA.

$$\frac{Time_B}{Time_A} = 1$$

$$\frac{Time_B}{Time_A} = \frac{(N*2.4*1)}{(N*CPI_B*1.5)}$$

$$CPI_B = 1.6$$

This means that in getting the same performance, we can change the CPI on processor B to 1.6.

Similarly, we can calculate the required changes to the cycle times and find out if identical performance can be obtained by changing the cycle time of processor A to 1.25, or alternatively changing the cycle time of processor B to 1.2.

Performance Estimation

As was previously noted, it is exceedingly challenging to develop a valid and widely accepted standard metre for assessing performance due to the diversity of computer systems (servers, desktops, laptops, tablets, smartphones, different appliances, etc.). Without such a standard, particularly in the era when computer systems were very costly, different suppliers created a wide range of measurements that may give them an edge. The key concept was to only provide a portion of the data, which resulted in a somewhat different image (see the section "Cycles Per Instruction-Based Measure"). For instance, the author worked on a benchmark* for a significant client in the late 1980s. Just two vendors took part in the event. According to the findings, vendor A was superior in terms of both performance and price/performance. The marketing personnel employed by the vendor of system B specified the benchmark results in a highly inventive manner due to its significance and potential impact on upcoming sales.

The performance of computer B was ranked second in a recent benchmark, whereas computer A's performance was rated one before last.

This real-world illustration shows that, despite the statement's complete truth, it was incredibly deceptive.

As a result of the sharp decline in computer costs, benchmarks are no longer conducted, but manufacturers continue to use a variety of inventive descriptors to attempt to distinguish their product. Smartphones, which provide a broad variety of capabilities in addition to being phones, are a clear example of this. As the majority of customers often do not need, for instance, a particularly high-resolution camera, this capability is employed as a sales gimmick. Several metrics (MIPS, MFLOPS, etc.) were proposed in an effort to offer a standard; for more information, read the section on "Processor's Internal Clock." Even these criteria, however, had serious drawbacks that the vendors might have taken advantage of. For instance, a computer with 100 MIPS can process 100 million instructions per second. The MIPS metric, however, does not specify which instructions are measured. Hence, even if it makes no sense, it is theoretically conceivable to choose the instruction that is quickest. It is possible to determine the speed of a system, for instance, by assuming that its quickest instruction is NOP (No Operation), despite the fact that this is an illustrative example that offers no insight into the actual performance of the system. The system's peak performance may be determined using these "standard" measures. Sadly, peak performance is a theoretical limit that the CPU will never reach. Since it is impossible to forecast the gap between peak

performance and actual performance, peak performance cannot be used to accurately predict how well programmes will perform in the real world. Yet when comparing the same software on two processors with comparable design, like Intel core i7* processors, peak performance might be helpful. The binary execution file will be same if, for the sake of comparison, the programme uses the same compiler, which results in the same amount of instructions. When the top performances of the two systems are known in such a situation, the performance of the second system may be better predicted by comparing the peak performance statistics to the actual performance recorded on the first system (by using the simple rule of three).

Assume, for instance, that system A's top performance is 50 MIPS and system B's peak performance is 60 MIPS. On system A, a certain programme was ran and reached 40 MIPS. We may assume that system B will run at a rate of 48 MIPS if system A and system B have the same hardware architecture.

There are many chances for businesses to provide new, impartial services as a result of the challenging scenario around the evaluation of system performance. Nevertheless, certain definitions are necessary before continuing.

Elapsed time: This is the length of time needed to execute the programme. This time comprises the time spent using the CPU as well as any additional time required for tasks like retrieving data from the hard drive or gaining access to memory. We cannot use the elapsed time to determine how much time the processor will require since it also comprises non-relevant components. Since it refers to the overall amount of time from the start of the programme to its conclusion, the elapsed time is also known as wall clock time. The period of time between hitting the Enter key and the results appearing on the screen is known as the elapsed time in systems engineering.

Processor (or CPU) time: This is how long the processor spent working on the application. As it excludes other time-related factors, this is not the reaction time. Certain operating systems, like Windows, don't track how much CPU time an application uses. Some operating systems that provide CPU time, like UNIX and Linux, occasionally distinguish between User CPU time and System CPU time, the latter of which refers to the amount of time the processor spent working on operating system tasks related to the application in question. The program's CPU time is calculated as the product of the system and user CPU times. Before the right choice could be made with the costly, older systems, a full analysis of their predicted performance was necessary. The staff of the computer department sometimes lacked the credentials needed for these examinations and simulations. In other situations, these standards came at a great cost. This prompted the creation of several benchmarking applications that provide a uniform framework for evaluating the performance of systems. This was offered as a service for a small portion of what actual benchmarks would have cost. Contrary to the mathematical manipulations of some of the manufacturers, the service supplied information on a range of systems, including in-depth comparisons with publicly accessible formulae.

Benchmarking initiatives

Throughout time, benchmark tools for evaluating system and processor performance changed. Based on the accumulated experience, each generation aimed to overcome prior constraints and provide more capability. The major goal was to create a collection of instruments that would provide the necessary data suitable to several computing disciplines. Validity and relevance of the outcomes are key considerations while creating such a collection of programmes. A software that mostly employs integer calculations, for instance, cannot be applicable to sites that typically use floating-point computations. A meaningful collection of benchmark programmes should also provide greater flexibility, such as a

parameter that specifies the split between integer and floating-point computations. Yet, there are times when even this flexibility is insufficient since the site is unable to provide its own blend. An interactive website that provides visitors with continually evolving programming is one example.

The early test programmes were tens of instructions long, practical benchmarks for small toys. Among these simple programmes were:

Hong Kong Towers Three towers, some discs, and this little game are used. The sizes of each disc vary. When all of the discs are stacked on one tower, they are ordered in size order, with the largest disc at the bottom. The stack of discs must be transferred to a new tower by the player; however, only the top disc may be moved, and it is forbidden to stack smaller discs on top of larger ones. Edouard Lucas, a French mathematician, created the game near the end of the nineteenth century. It is occasionally used as an exercise in the study of computer science recursion.

Eratosthenes' Sieve This method is used to discover prime integers. About 200 B.C., Eratosthenes created various algorithms (estimating the circumference of the earth, the distance to the sun and moon, etc.). His method of locating prime numbers is based on writing down all the numbers in a certain range and marking off any instances in which the prime numbers are multiplied. For instance, let's assume that someone wishes to locate all prime integers below n (we'll use n = 20 in this example for convenience). It will be necessary to start by writing down the whole range:

Algorithms of all shapes and sizes, and so on, make up a third category of benchmark programmes. All of the aforementioned benchmark programmes were real-world applications that were accessible. Running them on several platforms and getting the necessary metrics was simple. Sadly, the findings were only applicable to computer centres, which use a mix identical to the benchmark used for toys. For instance, the Hanoi Towers benchmark's findings were applicable to a mix that heavily relies on recursion and stack.

These inherent restrictions made it evident that more reliable benchmark test programmes were needed in order to offer information about the performance of the systems that was better and more reliable. The constraints of the toy benchmarks were overcome by the subsequent generation of benchmark programmes, which mostly consisted of synthetic programmes created specifically for the measuring procedure. These synthetic programmes have the benefit of being adaptable, allowing different parts of the processor's performance to be evaluated. With the programmes now in use in real life, this idea is not possible. Many benchmark systems have been established throughout the years, but we will only discuss two of them:

Whetstone benchmark: Harold Curnow created this software in the UK in the 1970s. The software was first created in ALGOL and used to evaluate the effectiveness of a computer system created in the UK at the time. Several programming languages were used to translate later versions of the software. Kilo whetstone instructions per second (KWIPS), a common unit of measurement used to compare the performance of various systems, were used to measure the program's outcomes. The majority of computations were made using floating-point math.

Reinhold P. Weicker created the synthetic software known as the Dhrystone benchmark in the 1980s. Dhrystone placed greater focus on integer arithmetic, procedure calls, pointers calculations, and other tasks than Whetstone did, which primarily evaluated floating-point arithmetic performance. The application was converted into C and quickly gained a lot of UNIX users' favour. The many UNIX-based workstations that inundated the computer

industry in the 1990s are what gave rise to its popularity. The number of Dhrystone loops per second is the outcome. Dhrystone MIPS was another another standard that resulted from employing Dhrystone (DMIPS). To do this, divide the total number of Dhrystone loops by 1757. The VAX computer, created by Digital Equipment Corporation (DEC), was rated as a 1 MIPS machine and produced 1757 Dhrystone loops while executing the Dhrystone benchmark.

In addition to the creation of synthetic benchmark programmes, there was a tendency towards employing bigger real-world programmes that may represent the workload at a location. Mostly for systems engineers, a benchmark was intended to be provided. Instead of investing time and money creating artificial programmes that are difficult to evaluate, the benchmark will mirror the behaviours of the users. While the synthetic algorithms take measurements of several system components, it is unclear how much of these components are relevant to the job at hand. One such use was the circuit simulation software SPICE.

The establishment of the Standard Performance Evaluations Corporation (SPEC) marked a major advancement in efforts to standardise performance assessment. This nonprofit company creates, maintains, and promotes the use of common measuring instruments. The firm, which is still around today, offered multiple iterations of benchmarking tools in response to advancements in computer technology. The programmes were to be made available to any vendor. The vendor completed the benchmark and returned the findings. The list that is accessible to the public was kept by the SPEC group.

Ten programmes that generated a measure known as SPECmark made up the first generation (1989) of benchmarking tools based on SPEC specifications. The use of 10 distinct applications causes the primary issue with this benchmark. There was no way to balance the two forms of arithmetic; some programmes mostly utilised floating-point arithmetic, while others primarily used integer arithmetic.

By establishing two distinct metrics for integer and floating-point computations, the second generation (1992) was created to get around the restrictions of the first generation's programmes. The SPECInt92 metric was created by 6 programmes that used integer arithmetic, while the SPECFp92 metric was produced by 14 programmes that used floating-point arithmetic. Vendors were not permitted to alter the applications' source code, but they were free to utilise whatever flags the compiler offered to maintain compatibility across different platforms.

The third generation, which debuted in 1995, included further improvements and alterations. The benchmark software was completely redone. Owing to the significance of the SPEC standards and their widespread acceptance, scores were a factor in many purchase decisions. As a consequence, several manufacturers changed their compilers and included different flags to better (faster) recognise the benchmark programmes. Regrettably, despite the fact that these findings were actual, they don't really indicate the performance that can be expected on the job site. The two kinds of metrics used in this generation were SPECInt95 and SPECFp95, which were generated using eight integer benchmark programmes and ten floating-point benchmark programmes, respectively. Running each application requires using the same compiler parameters. New versions of the benchmark programmes were produced in 2000 (CPU2000) and 2006 as they continued to develop (CPU2006).

The emphasis of the next generations was on distinctive and particular characteristics of computers. mostly as a result of the industry's quick developments and the variety of computer environments. As of 2014, SPEC offers hundreds of benchmark results for CPU, graphics, workstation, handheld, high-performance computing, client/server, mail servers,

and other components.

Parallel to the formation of SPEC, whose initial goal was to provide a common metric for processor performance, another organisation was established with the objective of assessing system performance as a whole. Another charity with a focus on defining and publishing performance metrics for transaction processing and database performance is the Transaction Processing Performance Council (TPC). The primary distinction between the two groups is that, whereas SPEC began by evaluating hardware performance, TPC focused on business activities. As the metrics are related to the processor performance as well as reading and writing the hard drive, database performance, and information transmission, this may be seen as a more comprehensive method. The TPC benchmark programmes show how mature the computer industry's processes are and how, despite the importance of processor performance, other potential bottlenecks must also be addressed to obtain optimum performance. Several benchmarking tools have developed through time, much like SPEC:

The first programme to be launched was TPC-A, which did so in 1989. Its primary purpose was to assess system performance in a basic online transaction processing (OLTP) setting. It was put to rest in 1995.

In 1990, TPC-B was created with the goal of assessing system performance in terms of transactions per second. Given that it represented the highest load the system could withstand, it was a stress test. It was also abandoned in 1995.

TPC-C was created in 1992, underwent further development, and is still in use today. The software replicates a sophisticated OLTP system with several databases and query types. TPC-C is used to recreate a real-world setting.

Other standards: Aiming at decision-support systems, which are often built on more sophisticated transactions and use complex data structures, TPC produced the TPC-D benchmark over time. In 1999, this benchmark was dropped. TPC-R, which was terminated in 2005, was designed for settings that generate reports. TPC-W, which was decommissioned in 2005, was designed for an Internet-based e-commerce environment.

In addition to TPC-C, TPC maintains a series of benchmarks for decision-support environments (TPC-DS), a brand-new OLTP benchmark (TPC-E), and others as of 2014. It should be noted that SPEC recognised the necessity for accurate measurements that represent the performance as experienced by the user and not just the processor's relative performance at the same time as TPC was established. In order to test different system setups, SPEC provides a large variety of benchmark applications.

Calculating the Results

The programmes are often performed many times while running a particular benchmark in order to remove certain external noise, such as delays in communication networks or operating system operations, etc. A lot of benchmarks also include a number of applications that each run for a varied amount of time. Thus, how to create a single average, cohesive, and particularly beneficial output after receiving many and occasionally varied times.

If the set of timing results from this sort of computation has a significant variation, a problem arises. In this situation, the lengthier periods have a tendency to impact the average and result in false findings. Consider the scenario where the majority of programmes at a company are tiny ones but a lengthy procedure occurs once a month. If an acceptable mix is to be created, the long term must be taken into account and reflected in the mix. Yet, including a programme with extremely distinct characteristics might lead to inaccurate findings.

For example, assuming the mix contains 99 programs, each one runs one second and one

program that runs 1000 s. The simple average that represents the small programs is one second however when calculating the combined average (including the long run), it changes to 10.99 s (1000 + 99 * 1) / 100 = 10.99

The result obtained is misleading because it does not provide meaningful information. This is an artificial number that does represent neither the short runs nor the long run. Weighted arithmetic mean that is defined by the formula,

$$\textbf{WAM} = \frac{\left(\sum \textbf{Weight}_i * \textbf{Time}_i\right)}{\sum \textbf{Weight}_i}$$

By giving a weight to each result obtained, this formula offers a technique for rectifying the distortion caused by the arithmetic mean. In the above example, the weighted average would be 1 s if the weight of each short run was 1 and the weight of the long run was 0.01. During the period of time that reflects the short runs, this is an accurate approximation. But, if the long run weight is set to 0.1, the estimated average will be 2 s, which again distorts reality. This example demonstrates how the weighted arithmetic mean has the ability to provide the desired outcome, but only if the appropriate weight values are chosen. Even when attempting to evaluate the relative relevance or frequency of the programmes, the approach is not straightforward. These weights often rely on the system in question and are difficult to adapt to new systems. When the same set of weight values must be utilised while comparing several computer systems, the situation gets noticeably more complicated. The weighted average mean can thus show to be less dependable and less objective.

The geometric mean, which is the product of the nth root of the findings (multiplying all the n timing results and then calculating thenth root). This mean is helpful when there is a significant amount of variation in the data, but it does not provide a reasonable indication of the true average. A geometric mean was used in the preceding example to get a value of 1.07, which is much better than the outcome of the arithmetic mean computations. In this instance, the variance is 9980. The arithmetic mean and geometric mean, however, will be 51 and 37, respectively, if we choose a set of 100 runs whose time increments by a constant—for instance, if the times are specified as “1, 2, 3, 4, 5... 100”—instead. In his instance, the difference is 842.

The fact is that it might be difficult to define an appropriate benchmarking capability. As each of the stated average approaches has certain intrinsic characteristics, not only do the programmes need to represent the real-life mix utilised by the site, but there is also a challenge in understanding the results.

CHAPTER 10

THE AMDAHL LAW

Shashikala H.K, Assistant Professor,
Department of Computer Science and Engineering,
Jain (Deemed to be University) Bangalore, Karnataka, India
Email Id- shashi.hk85@gmail.com

A phenomena that Gene Myron Amdahl characterized throughout the years became a pillar in assessing the performance of processors. Gene Myron Amdahl was one of the key architects of mainframe computers, notably the renowned IBM System 360. But, it may also be used in other fields, including general systems engineering.

According to Amdahl's law, a component's ability to improve performance is limited by the amount of time it is actually utilised. This equation is often used when estimating the performance gains to be made by introducing extra processors into the system or in multi-core contemporary systems. Yet, the legislation is not limited to computer-related situations; it may also be used in other contexts.

The following equation encapsulates the law:

Considering that FE is the percentage of time the improvement (or enhancement) may be employed PE is the performance improved upon, and the anticipated new execution time is provided by:

$$\textbf{Execution Time}_{\textbf{new}} = \textbf{Execution Time}_{\textbf{old}} * \left((1 - F_E) + \frac{F_E}{P_E}\right)$$

Using this formula, is it fairly simple to extract the speedup

$$\textbf{Speedup} = \frac{\textbf{Execution Time}_{\textbf{old}}}{\textbf{Execution Tine}_{\textbf{new}}}$$

$$\frac{1}{\left((1-F_E)+F_E/P_E\right)}$$

Assuming, for instance, that a programme has two sections, the first portion runs 70% of the time and the second part runs 30% of the time. The second half was rebuilt in an attempt to speed up execution, and as a result, it now runs five times quicker. How long will the programmes last overall when this adjustment is implemented?

Types of Processors

Despite the abstraction employed in processor design, which seeks to lower costs while retaining capabilities, a new generation of processors still needs a significant amount of financial resources to be created. As a result, manufacturers had to find methods to sell as many processors as they could in order to recoup their investment in this fiercely competitive industry. This is one of the primary causes for the demise of many old mainframe enterprises,

and the ones that are still in business employ commercial processors.

We must monitor the technical advancements during the 1990s in order to better grasp these trends. The number of newly forming computer enterprises increased in the initial years of that decade. Several created and offered UNIX-based workstations built on processors made with the "new" reduced instructions set computing technology (RISC; which will be discussed in the two next sections). The key breakthrough brought forth by RISC is the capacity to create and construct alternative processors that are fast yet easy to design and relatively inexpensive to develop.

Although there were dozens of these businesses in the 1990s, each one attempting to focus on a different niche of the market, IBM, which designed and produced the PowerPC*, which is now a component of the Power architecture but was initially designed to run UNIX-based systems, was the dominant player. Collaboration between Apple, IBM, and Motorola resulted in the creation of the chip. It was used in systems created and produced by the three businesses. However, Intel CPUs dominated the primary personal computer industry, making it difficult for the new chip to gain a significant market share. Nonetheless, until 2006 (when Apple moved to Intel's CPUs), the Apple Macintosh system utilised the chip. The chip is now primarily used in a variety of high-performance embedded appliances.

Minicomputers were first popularised by the successful computer business Digital Equipment Corporation (often known as DEC or Digital). In 1998, Compaq (a major manufacturer of personal computers and systems compatible with IBM PCs) purchased DEC. Alpha is a quick 64-bit semiconductor that DEC created and produced before that purchase. There were no substantial partnerships with other computer system makers, and DEC systems dominated its usage. This prevented other manufacturers from sharing the expensive design expenses, which may have contributed to DEC's financial difficulties. HP purchased Compaq in 2002, however the brand name is still used, mostly for entry-level computers. Before to then, the Alpha technology and the associated intellectual property rights were sold to Intel since Compaq was an Intel client, thereby ending the technology.

A series of processor chips made by MIPS Computer Systems were created with both commercial systems and diverse embedded devices in mind. SGI (Silicon Graphics, Inc.), a manufacturer of high-performance 3D-graphics computer systems, employed the company's CPUs. As a consequence, SGI bought MIPS in 1992. Sadly, SGI opted to move to Intel CPUs as the system's engine a few years later, and MIPS was spun away. It was purchased by the UK-based manufacturer of embedded CPUs, Imagination, in 2013.

A further successful business that helped shape the computer sector was Sun Microsystems, Inc. A key contributor to the development of some of the most widely used technologies today, including UNIX, Java, the Network File System (NFS), Virtualization, and others, was Sun Microsystems. Despite its valuable contributions, Oracle Corporation purchased Sun Microsystems in order to provide a high-end integrated (hardware/software) solution designed for big businesses and cloud-based systems.

The x86 series of microprocessors, which is at the centre of the majority of 32-bit personal computers in use today, was developed by the semiconductor manufacturer Intel (see Table 1.2). The personal computer sector's rapid technical advancement and Intel's involvement in the hardware market contributed to its spectacular success. Intel established itself as the top CPU provider throughout the 1990s despite having a number of rivals, including AMD.

Founded in 1939, Hewlett-Packard (HP) is a producer of computer hardware and accessories. Despite having its own brand of CPUs, the business chose to collaborate with Intel to create a new range of processors (Itanium). The hardware design and manufacturing businesses

realised early on that obtaining high volume sales required strategic collaboration with other manufacturers. These sales' earnings could cover the substantial expenditures incurred by technological advancements. This is the rationale behind the majority of chip makers' strategic alliances and collaborations in the 1990s.

The RS/6000 used IBM's PPC, which was also included into Motorola's systems and computer equipment. It was used by Apple until 2006 and is also utilised by IBM in several of their controllers. Silicon Graphics and a number of game console manufacturers both utilised the MIPS processor. The tremendous demand for these systems allowed Intel and other personal computer processor makers to benefit from lucrative income streams.

Only DEC failed in its attempts to entice business partners who might provide the money needed for future improvements. This may have been a red flag for the company's capacity to go forward and succeed. Compaq computers, a company that made mostly personal computers, bought DEC in 1998. Not the Alpha microprocessor, but DEC's global marketing, distribution, and support network was of more worth. Sadly, the united Compaq-DEC firm could not survive, and HP purchased it in 2002. The major reason why this short historical account of processor developments over the last two decades is significant is because of the lessons it may teach. Even big, successful firms nowadays must always come up with new solutions due to the rapid growth of technology. The only way to recover the investment made in these new advancements is to increase market share and via strategic alliances and collaboration. These collaborations are now much more essential due to the even quicker technical advancements, the ever-changing market, and the implementation of new standards in the sector.

There were two technologies utilised to create processors in the 1990s:

The majority of personal computers and mainframes used the sluggish and expensive CISC (Complex Instructions Set Computer) technology. Nonetheless, because of their mass production, personal computer processors were affordable. Even when compared to the massive mainframes, the newly developed (at the time) UNIX-based workstations utilised RISC technology, which was substantially quicker. Although being more straightforward, the technology was nonetheless pricey since there were less workstations than PCs.

The RISC technology was implemented by Intel and other PC processor makers by the 1990s' end, which decreased its cost and improved the performance of personal computers. Because of the developments in the computer industry, the processors' map throughout the second decade of the twenty-first century is obviously different. Every year, more portable devices (such as smartphones, tablets, and other gadgets) are sold than personal computers, and each of these new devices has a particular processing need. Because of this, new businesses have emerged that provide processors that are more suited to the demands of the modern market. For instance, modern CPUs have reduced heat dissipation and energy usage.

The RISC contribution to performance improvements in the computer industry is explained in further detail in the following two sections.

Technology CISC

Throughout the early years of computers, the CISC technology predominated. A simple method of creating and implementing additional new instructions was made possible by the development of microinstructions (see the section "Performance" in this chapter). These directions were not all that simple. At the time, it was accepted wisdom that more complicated instructions would facilitate the compiler's task of converting high-level programming instructions into machine-level instructions. The microinstruction approach worked well since it was obvious that programmes would get more complicated over time

and that developers would require more durable instructions. Because of this, several manufacturers included complicated and perhaps pointless instructions in the belief that doing so would benefit programmers and compilers. In actuality, these intricate instructions offered higher-level programming languages greater support. The CISC-based computers featured hundreds of distinct machine instructions and several addressing modes since writing new instructions was generally straightforward. One of the key factors in the complexity of the instruction is these addressing modes, which specify how the instructions address memory. A brief list of some of the addressing modes utilised by CISC-based computers may be found in Table 4.6.

These addressing modalities have the following meanings:

Register: Only registers are used in the instructions when this addressing method is used. In this example, the instruction adds the data in registers R5 and R3, then stores the result in R5. This is an illustration of a register-register-based architecture; for more information, check the section under "Architecture Synopsis."

Immediate: Instructions that employ a register and an instantaneous value that is stored inside the instruction itself use this addressing style. In this instance, the value in R2 is increased by the constant 3, and the result is then inserted back into R2. It's crucial to note that commands with immediate values take up more space (in terms of bytes in memory). While the developer might utilise huge instantaneous values that would need more bits, the number 3 in this particular instance is tiny enough to fit even in 4 bits (half a byte). As a result, these instructions will either need to utilise variable lengths or an alternate preset length that may constrain the values used. For instance, it will restrict the number of values that may be utilised in such an instruction to 1024 values if one byte is set aside to keep the immediate value.

Displacement: With this addressing style, even when instructions are carried out across registers, the register may be enclosed in parenthesis, in which case it serves as a pointer (the value in that register is the address in memory). The instruction also contains a displacement that should be added to the register address. In this example, R1's content is raised by 100 (the displacement), and the resultant value is the memory address. The contents of R3 are then supplemented with the contents of this address, and R3 is then saved with the result. The displacement lengthens the instruction similarly to the prior addressing style.

Subtract indirect: This is a more straightforward addressing method than the one before it. Without the ability to add a displacement, the second register is utilised in this instance as a pointer. R1's data contains a pointer to a particular memory address. The contents of R2 and this memory location are then combined, with the result being stored in R2. This instruction is shorter than the last one since there is no displacement in this addressing mode.

indexing on: employing registers to carry out instructions, some of which serve as pointers. A pointer is a register that is enclosed in parenthesis. With a minor addition, this addressing method is comparable to the preceding one. The sum of the contents of two registers may be used to compute the pointer. In the illustration, the content of R1 and R2 are added, and the result indicates a memory address. The contents of R3 and this memory cell are combined, and R3 then stores the result. By offering a method for managing tables and two-dimensional arrays, this addressing mode expands on the prior addressing style. Whereas the second register contains the address of the field, the first register has the address of a particular row in the table (the displacement in the row).

Direct or absolute addressing: This addressing style permits operations between registers or between registers and memory locations. The contents of cell 1001 and R3 are combined in

the example's instructions, and R3 is then used to store the result. The instruction will need to be lengthier, just as in earlier situations when the instruction had a constant value. This is an example of a memory-register-based architecture (see the "Memory-Register Architecture" section for further information).

Memory oblique: In this addressing method, operands from memory and registers are used to carry out instructions. In the illustration, the instruction adds the contents of R1 and a memory location whose address is in R4. Similar to earlier situations, R1 stores the amount.

Auto increment: This addressing mode extends the scope of earlier modes that were previously established. At the area indicated by R2, one operand is present in memory. R1 contains the second operand. The instruction sums the two figures, and R1 stores the result. Moreover, R2's value is automatically increased by the instruction such that it points to the subsequent value. Because there is no longer a need to manually increase the pointer, this is incredibly helpful for a loop. This is an example of an instruction and the corresponding addressing mode intended to enable instructions from higher-level programming languages.

With a little exception, this instruction is extremely identical to the one before it in terms of functionality. The auto decrement lowers the value of the register whereas the auto increment raises it (advancing the pointer) (moving the pointer backward).

Scaled: This addressing style is more intricate. It is used to combine the contents of a register and a memory location. In this scenario, the content of R3 multiplied by a constant, together with the value of R2, and the displacement of 100, provide the memory address. The purpose of this addressing mode is to accommodate more complicated data structures, and as in earlier instances, this means that the instruction will need to be longer in terms of memory use.

Using CISC technology made it easier to translate commands from high-level programming languages into machine instructions and gave developers the tools they needed to create compilers that were less complex. The code was more compact thanks to these intricate instructions, which lowered the program's memory requirements. This was a valuable feature when memory was quite expensive. This was not advantageous at all since memory prices dropped significantly. On the other side, CISC causes some significant issues, both financially and in terms of impeding the rapid technical progress that is needed.

Two main components make up the CPU, as was previously mentioned. the CU, which is in charge of control and scheduling, and the ALU, which is responsible for carrying out the instruction The CU retrieves the instruction from memory, decodes it, and then brings the required operands before sending it to the ALU for execution. As the next instruction cannot be started until the preceding instruction is complete, the CU for CISC must be familiar with all the instructions, the different addressing modes, and the amount of time required to execute each instruction. While CISC offered certain benefits at the time, the CU itself became exceedingly complicated due to the complicated structure of the instructions and their addressing modes. The CU contained the majority of the processors' logic, and each new instruction that was put into use added layers of complexity. Even the length of the instructions contributed to some degree of complexity in terms of memory utilisation. Although instructions that have an immediate operand (that is stored as part of the instruction) must be larger, instructions that employ registers as operands may be relatively brief (2-3 bytes only). Moreover, certain instructions, like the Scaled instruction, have two immediate operands and must thus be much longer by definition. As a result, the CU must determine how many bytes are required by each instruction. The CU begins reading the instruction, decodes it, and only after the decoding step is complete does the CU learn the

format and the number of bytes needed for the instruction. When the necessary operands are ready, the same procedure is repeated. In circumstances when the operands were in registers, the fetch was easy and uncomplicated. The operands were in memory at other times, forcing the CU to out where they were. As the CU had to be able to do simple calculations as well, this went against the initial theory that the ALU performs the computations while the CU manages the operations. It turned out that the CU contained the majority of the logic and complexity of the CISC processors. The intricacy of the problem showed itself in greater design costs for new CPUs, which slowed down technological advancement. These issues led to the creation of a new computer technology called RISC.

RISC Technology

RISC technology has been considered for a very long time, and certain mainframes from the 1960s made use of it to some extent. Yet it didn't really take off until the 1990s. The challenges with CISC and the realisation that the sophisticated CISC instructions were seldom utilised served as the primary impetus for implementing this "new" technology. Additionally, it was unclear whether they were truly quicker than a loop of simpler instructions owing to the cost involved by these sophisticated instructions.

Compilers must be more advanced in order to optimise code and make creative use of the registers. The notion that software offers a far quicker development path was what drove the RISC technology ahead. While software engineers have inherited several of the ideas employed by hardware designers, such as modular design and simplicity, it is often quicker to deal with some of these problems with software. It was time for a change once it became obvious that the CU had become sophisticated and was slowing down technological advancement. The architecture, particularly the CU, was made simpler because to new advancements in software engineering, which also increased the flexibility and sophistication of the compiler. More instructions are produced per programme when RISC technology is used. This implies that additional memory will be needed by the software. Nevertheless, particularly in terms of the CU, the CPU architecture is more straightforward and less expensive. A pipeline execution method is made possible by the consistent execution times (as will be explained later). The ability to execute one instruction per cycle is made possible by the pipeline idea, which is a significant performance enhancer. Regrettably, there is a disconnect between the intended ideas and their actual execution for a variety of legitimate reasons. When certain instructions are really basic, like IF or integer ADD, while others are more difficult, like DIVIDE, it is impossible to have identical execution durations. To solve these issues and get the same outcome, lengthier running instructions were divided into many pieces.

Machine instructions are constructed from components used in regular execution, as was previously mentioned (see the section "Performance") (microinstructions). The ability to incorporate new capabilities and new instructions is made possible by integrating these microinstructions. RISC technology also makes use of these basic components.

the two primary parts of the CPU. The ALU is shown on the left as having three (or more) registers to store the operands and the result, among other things. The CU transfers the operands from their location (register, or memory) to these internal registers before executing the instruction. Using unique input buses, the ALU obtains the operands before obtaining the particular operation to be carried out. The obligation for copying the result to the destinations specified by the instruction rests with the CU once the result of the instruction has been completed and saved in the special destination register (by means of an output bus).

The microinstructions (or execution phases) used by various RISC implementations may

vary, but typically they should include: Determining the address of the next instruction to be performed, Taking the command from memory, Decoding the instruction to ascertain its validity, the number of operands, and their kind calculating the operands' addresses obtaining the operands, carrying out the directive, calculating the address of the outcome, transferring the outcome from an internal register to a different register or memory location

These phases, sometimes known as microinstructions, may be enhanced by certain processors, while others may break some stages up into smaller stages. However, these stages are a crucial component of instruction-level parallelism (ILP; this will be explained in this chapter).

CISC and RISC

We will examine the execution of a particular instruction and how it is carried out by each technology in order to compare CISC and RISC and highlight the benefits and drawbacks of each. The memory stores the data and the instructions, while the CPU executes the instructions, according to the von Neumann architecture, which laid the groundwork for contemporary computers. Every place in memory has an address that may be used to access the data there. The CU accesses memory; the ALU does not; the ALU only executes the instruction after the operands have been put in internal registers. Let's say we need to multiply two integers, one of which is at address 1058* and the other at address 2076. Address 2076 will be used to store the outcome. Because of the (at the time) high prices of memory, the CISC technology was created to reduce the amount of instructions and, as a consequence, the programme as a whole would use less memory (tens of thousands of dollars for 32 KB). This indicates that complicated instructions were supported by the hardware during design. In this straightforward example, the compiler will use the MULT instruction to first load the two operands into the ALU internal registers. After the multiplication is complete, the result, which is saved in another internal register, will then be placed in the appropriate position in memory (2076 in this example).

As the CU transfers the operands from memory to the internal registers, the instruction uses memory resident operands, making it a straightforward translation of an equivalent instruction found in high-level programming languages. The instruction as described is equivalent to the following instruction if, for instance, the variables at positions 1058 and 2076 are designated A and B, respectively:

First impressions lead one to believe that the RISC implementation wastes memory since it uses four instructions as opposed to the CISC version's single instruction. Also, the compiler will have to put in more effort during compilation owing to the disparity between the RISC machine language instructions and the higher-level instructions. The RISC implementation does provide some additional key benefits, however. The majority of the instructions' comparable execution times make it possible to build a pipeline. By leveraging ILP, such a pipeline has the ability to accelerate execution (Instruction Level Parallelism will be explained in the following section). Because of this, it's feasible that the four RISC instructions will need almost the same amount of time to execute as the single CISC instruction. Yet, the RISC implementation requires fewer resources and takes less time to complete, which speeds up the development of future CPU generations. However, even without technical reduction, the simpler CU suggests that fewer transistors will be put on the device, creating room for extra registers. Indeed, the processor's job is made simpler by separating the LOAD and STORE operations from the execution itself. The CU had to load the internal registers for every instruction in the first CISC implementations, even if the value was already in one of the registers because of the prior instruction. This results in an extra, pointless memory access. Values are stored in registers in the RISC implementation and

remain there until a different value is consciously substituted for them.

As can be seen, there was no need to load the variable A again since its value in register R1 remained constant. Regarding the "Iron Law," the CISC technology attempts to reduce the amount of machine-level instructions while disregarding the number of cycles needed for each instruction (see the section "CPI-Based Metric"). Contrarily, the RISC implementation ignores the growth in the number of instructions in favour of trying to boost the CPI.

Despite the RISC technology's built-in benefits, it took some time before it was extensively used in commercial systems. The absence of software appropriate for the technology was one of the causes. With the use of advanced compilers that tailor the code to the architecture, significant performance advances have been made. The expected advantages of using a nonfasted compiler will not be realised.

Parallelism at the Instruction Level

The fixed duration execution instructions, which allow parallelism at the single instruction level, are one of the key tenets of RISC technology. The pipeline mechanism, which divides the single instruction into its microinstructions and executes them in parallel, enables the parallelism. Each instruction, as indicated, is divided during execution into a number of separate steps. Calculate the address of the instruction. When the instruction takes up a different amount of bytes or when the preceding instruction was a branch, this is required. If: The Processor receives the instruction through an instruction retrieve from memory. ID: Directive Decoding includes figuring out the operands of the instruction. Determine the operand addresses. It's conceivable that the instruction accesses certain memory resident variables after being decoded. The addresses of these operands are determined at this step (Figure 10.1).

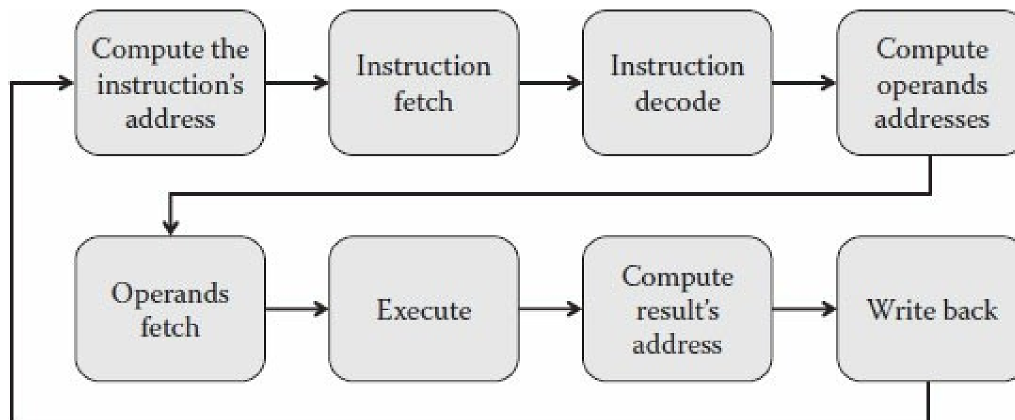

Figure 10.1: Instruction Level

Assuming that each of these microinstructions runs in a single cycle in the conventional (serial or nonpipelined) execution mode, it will take four cycles to finish each instruction (as seen on Figure 1). Each of the five instructions is broken down into the four phases described above. The first instruction will be processed over the course of four cycles before the following one begins. It will also take four cycles to finish before the next instruction begins, and so on. The five instructions (in this particular case) take 20 cycles to perform in total. The attempt is to have constant duration microinstructions, as indicated in the preceding example, since RISC technology aims to have equal execution durations for instructions, but sometimes objectively it is impossible. By using RISC technology, the hardware is created in a manner that allows each step to be carried out by a distinct CPU component (or unit). Such units should not share resources with other units and each should be self-sufficient. This

indicates that the instructions may be carried out concurrently. Like in the preceding (normal) situation, the first instruction begins to run and takes four cycles to finish. The instruction is read from memory in the first cycle. The instruction travels through the decoding step during the second cycle. Nevertheless, since the decoding is carried out in a separate independent hardware unit, the fetch unit is not in use during this second cycle. As a result, even if the first instruction is still being processed, the fetch unit begins fetching the second instruction instead of sitting about waiting for its turn. This indicates that the CPU is decoding the first instruction and fetching the second instruction during the second cycle. The next cycle also uses the same reasoning. The first instruction is being carried out in the third cycle, the second instruction is being decoded, and the third instruction is being acquired from memory. At any given moment, four instructions are being performed in stepwise parallelism since the execution of the instruction in this case was divided into four parts. It should be emphasised that this parallelism does not alter the execution time of a single instruction, which in the example above stays four cycles, but it does provide a method through which the CPU performance is improved. The previous illustration makes this point quite obvious. The pipeline approach reduced the number of cycles needed to complete the five instructions from 20 to only 8. The ratio between the initial execution time and the improved (pipelined) execution time may be used to describe the performance increase (or speedup). The speedup in this particular instance is (20 / 8 = 2.5).

Issues with Instruction-Level Parallelism

The execution time of a single instruction is not improved by ILP, but this is not a concern since there is no point in executing a single instruction. What counts is how long it takes for the application to execute, which involves carrying out millions of instructions. The number of pipeline stages represents the theoretically achievable speedup, but since these stages aren't always balanced, the maximum speedup isn't always achieved. When all stages of a pipeline complete their tasks at the same time, the pipeline is said to be balanced. There are three steps, and they are carried out in the order left to right. Every level requires a certain amount of time (10 ns).

An imbalanced pipeline is one in which the amount of time needed to complete each step may vary. The phases are carried out in the same order as in the preceding image, from left to right, but each stage requires a different amount of time than the others. In this scenario, the time must be specified as the lengthiest step in order for the pipeline to function correctly. This indicates that even if the first stage ends in 5 ns, the stage will remain inactive for an additional 10 ns until the second stage ends. The third stage also exhibits the same behaviour, however in this instance there will be an extra 5 ns of waiting time. Unbalanced stages in the pipeline are an example of a difficulty that restricts the potential speedup; as a result, these issues are often addressed by the designers of the hardware systems. Adding phases to the scenario is one straightforward fix. The three stage unbalanced pipeline may be converted into a six stage balanced pipeline by splitting the second stage into three 5 ns phases and the third stage into two 5 ns stages.

The internal clock of the processor was previously discussed (see the section on the "Iron Law" of processor performance), but the explanations and purpose were focused on the internal clock that synchronises the processor. Each system consists of a number of clocks (at least two) that operate at various frequencies. The processor cycle is determined by the internal clock, which also influences how quickly instructions are executed. The external clock establishes the rate of communication between the CPU and other external components, such as memory or input and output devices. The overall design, but the devices impacted by the two or more distinct clocks are shown.

For instance, the memory speed on an Intel Atom CPU C2750 is 1.6 GHz, whereas the processor itself operates at 2.4 GHz. This is due to the fact that the internal clock must be faster because it is used to synchronise the instructions or portions of the instructions, whereas the memory is an external component that, despite being a part of the system, is not a processor according to the von Neumann modular architecture. There are various steps involved in the process of retrieving data from memory. Initially, the necessary data must be "brought" from memory via the memory controller. The relevant data is located in memory by the controller, which then passes it to the CPU. Buses, which will be discussed in more detail in the bus chapter, serve as pipelines that transport data between the different devices and are used for all data transfer.

One of the motivations for implementing RISC-based systems was the speed gap between the processor and the memory, which causes delays in the processor's work every time data needs to be brought or sent to the memory. RISC architecture emphasises heavy register usage in an effort to minimise memory accesses. Nonetheless, it should be emphasised that RISC architecture, although having several registers, cannot completely eliminate memory access. When it occurs, the instruction flow is often stopped, increasing the execution overhead and slowing down (as will be explained in the following section).

CHAPTER 11

RISKS OF INSTRUCTION-LEVEL PARALLELISM

C R Manjunath, Associate Professor,
Department of Computer Science and Engineering,
Jain (Deemed to be University) Bangalore, Karnataka, India
Email Id- cr.manjunath@jainuniversity.ac.in

ILP risks are unique dynamic situations that make it difficult for the pipeline to run efficiently. There may be instances in software development when an operand from a prior instruction is used in a subsequent instruction. This may also occur when translating a single instruction from a higher-level programming language into many machine language instructions. Assuming, for instance, that A, B, C, and D are all programme variables and that the developer typed the higher-level instruction:

$$A = B + C + D$$

Since in most systems the machine instruction ADD accepts only two operands, the compiler will have to split the higher-level instruction into two machine instructions.

$$R0 = R1 + R2$$
$$R0 = R0 + R4$$

The compiler will need to load the variables B, C, and D into registers before the ADD instructions, for example, R1, R2, and R4. There is no issue with CISC-based systems or systems without a pipeline since the second instruction only begins execution after the first instruction is finished. This is a challenge for pipelined systems, where the instructions are carried out concurrently. As R0 represents the outcome of the prior operation, which is still running, the second instruction will have to wait when attempting to acquire the first operand (R0). Considering a four-stage pipeline (as explained in the "Instruction-Level Parallelism" section): The first instruction is fetched in the first cycle. The first instruction is decoded and its operands (R1 and R2) are fetched in the second cycle. The second instruction is fetched concurrently. The first instruction is being executed in the third cycle, while the second instruction is being decoded in parallel and an effort is being made to retrieve its operands (R0 and R4). The initial instruction that determines R0's value is still running, thus even if R4 is accessible, its contents are incorrect.

This implies that in order to retrieve the necessary operand, the pipeline must halt and wait for the first instruction to finish. Even if the hardware may sometimes overcome this kind of risk, it is preferable if the developer is aware of how the hardware functions and the effects of the high-level programming directives.

Data Hazards

The earlier-mentioned issue is a particular instance of data dangers. These risks result from the absence of necessary data in a timely manner. There are typically three types of data hazards:

When a later instruction tries to access an operand that was computed by a previous instruction but the operand is not accessible because the preceding instruction has not yet written it, read after write (RAW) happens. A RAW danger is the one from the preceding section. When an instruction reads an operand after a subsequent instruction has already altered it, this is known as write after read (WAR). For instance

It seems to be impossible for the first instruction to retrieve R1 once the subsequent instruction updated it. Execution out of order is one of the methods current processors employ, as will be discussed later, to speed up execution. This risk will manifest if the approach is used in this situation and the second instruction is carried out before the first one. Moreover, if the pipeline is not balanced, the microinstructions employed may take longer to complete (see the section "Level Parallelism Issues"). There are certain instructions that take longer to execute than others, such as the DIV (divide) command as compared to ADD. Under such circumstances, the execution step will often be divided into many portions in an effort to balance the pipeline.

The pipeline in this instance contains five phases, although the execution stage could need a number of extra cycles. Although the floating-point ADD instruction is more straightforward to execute and has just two execution segments in this theoretical system, the floating-point MULT (multiply) instruction has four one-cycle subsegments. If the example is significantly adjusted to include, for example, Most of the time, the issue must be avoided by the compiler. By altering the currently being used registers, this is simply accomplished. As the RISC architecture implements several registers, there is little overhead when replacing utilised registers with new registers. The code will change as a result of replacing R1 with another register, such as R8, in this situation:

$$\mathbf{R3 = R1 * R2}$$

$$\mathbf{R8 = R5 + R4}$$

Due to the different register to be used, this solution is sometimes called register renaming.

When a later instruction is writing its result and a prior instruction affects it, write after write (WAW) happens. It is not logical to express it in this manner since it suggests that two instructions are altering the contents of the same register. But, if it does, the hardware will detect and ensure that it runs properly.

An example may be

The compiler is accountable for addressing this risk, as was the case with the last one.

Access to Resources Conflicts Risks

Conflicts over access to different resources are a typical risk that affects the pipeline's performance (or pipeline stages). By providing distinct and independent execution units (resources) for each pipeline step, the ILP was made feasible. Most of the time, there will only be one unit per step, meaning that if there are five stages in a pipeline, there will be five distinct types of these resources. When two instructions try to access the same resource within the same cycle, access conflicts may occur. As an example, we shall examine how the following directions were carried out:

We will construct an execution table in which each column is a cycle and each line denotes a single instruction in order to assess the execution of the machine instructions. We will employ a four-stage pipeline to keep things simple (fetch, decode, execute, and write back). When no risks arise, the pipeline is performed without interruptions according to the regular and standard execution.

Sadly, the code provided does not reflect this. The operand used in the second instruction is not available when it is needed (data hazard). Thus, the first instruction is performing the ADD operation in the third cycle, the second instruction is in the decoding stage and attempting to retrieve the operands, and the third instruction is in the fetch stage. The issue occurs when the second instruction must attempt the fetch in the cycle after it fails the first time because it is unable to retrieve one operand (R0) since it is not yet available. This indicates that the presumption that each step would be completed in a single cycle is unfounded. The ID unit will have to wait for another cycle before making another attempt to acquire the necessary input operand. This indicates that the ID stage will take more than one cycle, at least for the second instruction.

Nevertheless, the execution of the third instruction is also affected by the data risk incurred during the execution of the second instruction. The third instruction in the fourth cycle was meant to be in the second stage in a typical (hazardless) scenario (decode the instruction and fetch its operands). Nevertheless, since each stage only has one unit (resource), and because the second instruction is still using the decoding unit, the third instruction is unable to complete the third step and must wait one cycle before attempting again. In this particular instance, things are really worse. Only when the first instruction is finished will the input operand needed by the second instruction be accessible. This indicates that the operand will need to be obtained by the second instruction after waiting for an extra cycle. Due to the need to wait until the decoding unit is accessible, this also influences how the third instruction is carried out. It depicts the circumstances in the sixth cycle.

Conflicts over accessing the few resources led to a delay in the third instruction's execution (the single decoding unit). It should be noted that the operand fetch was carried out as part of the decode cycle in this example of a four-stage pipeline. In certain implementations, acquiring the operands may be done during the execution phase. In these situations, the issue still remains, and the data risk will show up by causing the execution stage to be delayed and the conflict to be on the execution unit. If there are no dangers, the pipeline mechanism functions well and improves performance significantly. Regrettably, these risks can always remain and hurt how the pipeline operates. In the preceding example, the second instruction's data risk led to the third instruction's resource conflict risk, and if there had been any more instructions to be performed, it would have impacted all of them.

These dangers make it simple to compute the punishment or execution delay. On a nonpipelined architecture, these three instructions must be processed in twelve cycles. The pipelined execution of the three instructions should have taken six cycles under ideal and typical conditions. The time is cut in half, which is a major improvement. Unfortunately, because of the risks, the execution consumes eight cycles, which drastically reduces the pipeline's capacity improvements. The actual improvement was just 30% as opposed to the potential improvement of 50%. Owing to the execution time penalty brought on by the numerous dangers, which may sometimes be significant, steps were made to mitigate or lessen the penalty's effects. The original plan was to break the ALU into distinct functional components. As a result, there will be units for things like ADD, MULT, DIV (division), and so forth. Even though there is only one ALU, it has been split up into many functional components. Hazards may still arise in certain circumstances despite this divide, however. Hence, the establishment of several functional units came next. Two decoding units might lessen the detrimental effects in the resource conflict hazard case. This kind of thinking let people realise that a system can be built to do the tasks it must. Several floating-point units, for instance, may be advantageous for a computer system designed primarily for floating-point calculations.

The fundamental premise is that the Instruction Fetch unit is fast enough to handle the two pipelines concurrently without adding any latency. The CU must move the execution of a particular stage to the next functional unit when it is finished. It typically employs a simple flip-flop mechanism in which two functional units that are identical to one another are assigned alternately. For instance, all instructions will be routed to unit zero for odd instructions and unit one for even commands. This implementation uses a static allocation. There is no method to switch to the alternative pipeline once a pipeline has been assigned for an instruction. Another option is to create a dynamic allocation in which the instructions are distributed dynamically at each step. In contrast, the design takes a different tack. It is a five-stage pipeline system in this instance, just as in the preceding one. Nevertheless, just the ALU functionality was split up into several functional parts in this case. In this example, it is assumed that all other functional units—aside from the ALU—are quick enough to maintain the system without any hiccups. This illustration applies when the bottleneck is brought on by the execution (in the ALU) rather than the CU's preparatory steps.

Adaptive Scheduling

The RISC-based pipeline was implemented in several systems decades before the 1990s, when it became generally acknowledged. A technology like the contemporary RISC was used by the American computer corporation Control Data Corporation (CDC), which operated in the second half of the 20th century. Seymour Cray created the pipeline-based CDC 6600* computer at the start of the 1960s. The CDC computer used two different phases in place of the Instruction Decode step seen in RISC technology. The first was used to retrieve the operands, the second to decode the instructions and wait (if there are any access conflicts). Its design was primarily motivated by the notion that access conflicts and collisions are common during parallel execution and that a stage that would wait and synchronise the activities is necessary. Ten functional units were employed in the CDC 6600's ALU. As compared to other computers at the time, these 10 functional units were one of the key factors in the machine's execution speed (1964). the RISC-decoding cycle was split into two distinct stages: Stage I of the process, which decodes the instruction, finds the necessary operands, and determines if there are any access issues (data or access conflicts hazards). If one or more of these hazards were discovered, the stage will start a delay that lasts one cycle. A fresh check will be carried out in the next cycle, and so on until there are no risks and it is possible to retrieve the operands. After all operands are ready, the execution may begin. Many processors still use the 1960s-era technique for determining the operands' availability, which will be discussed in more detail in the next section. Stage R, where the operands are fetched. Because the previous step took care of checking their availability, there is no need to do so now. If the operands weren't accessible, the previous stage would still be waiting. The execution unit receives the fetched operands. The CDC 6600 also used dynamic scheduling, which was designed to reduce certain risks and speed up execution, in addition to the unique pipeline. Dynamic execution refers to carrying out the commands in a different sequence from the one in which the developer wrote them. It should be emphasised that the hardware implemented this method without any help from the developer or compiler. For this reason, the system should incorporate complex processes for assuring the proper outcome notwithstanding the modifications in addition to the hardware algorithms that decide on order changes.

Memory

A fundamental and crucial component of the system is the memory. A programme cannot be run without a memory to store its instructions and data, just as instructions cannot be executed without a processor. The original computers could only execute one programme at a

time since the memory and CPU were completely connected. It took a lot of time and effort to load a programme into memory. The computer can only run a programme after it has been loaded into memory. A programme must still be loaded (at least partly) into memory in order to function at the present time. Yet, as stated by the von Neumann design, memory is a separate modular component in contemporary systems. The greatest size of the programmes that could be executed and the number of programmes that could run in parallel on the first computers were both constrained by the comparatively minimal memory that they could support.

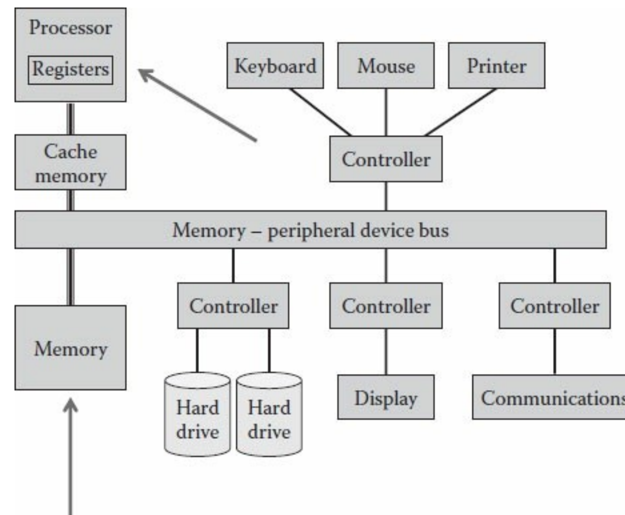

Figure 11.1: Memory architecture

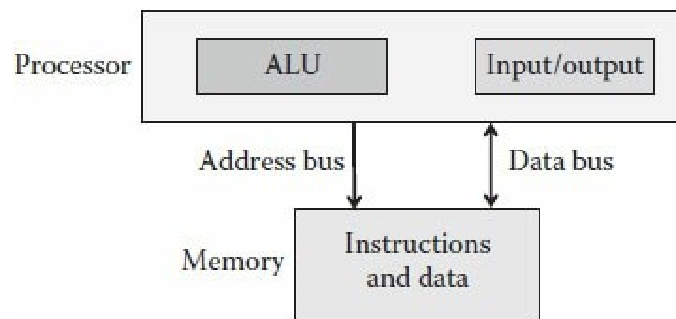

Figure 11.2: Von Neumann architecture.

As previously mentioned, the advent of the Von Neumann architecture marked a significant advance in the field of computing. The design (Figure 1) separates the system into a number of distinct components so that changes to one component in the future will have a minimum impact on other components. While many other sectors today use this kind of modularity, it was von Neumann's theories that made it viable for computers. Before von Neumann, the logic of the computer was closely interconnected, and any changes made also had an impact on other components. In terms of a different industry, it is comparable to a driver who has to change an engine after changing the tyres on a vehicle. Computer systems' introduction of the von Neumann-inspired modularity made it possible to minimise complexity and allow quick technical advancements. For instance, expanding the memory's capabilities and capacity was

simple to accomplish without requiring any changes to the other system components (Figure 11.1 and Figure 11.2).

The capacity of computer memory was increased, which was one of the most significant advances and led to greater system use. By allowing the central processing unit (CPU) to execute many programmes simultaneously, the utilisation was increased. The CPU would not have to sit idle while waiting for I/O operations to finish during the input and output (I/O) operations of one application. As there is memory for several processes, the CPU may be kept active even if one or more are waiting for input or output.

The necessity to provide a standard method for data transfers between the various components was one of the effects of breaking the system up into independent components. The device is known as a bus, and it will be described in more detail later in this book. The bus is in charge of sending instructions and data from the memory to the CPU as well as transferring part of the data back to be stored in the memory when it comes to memory-related tasks.

It should be emphasised that instructions and data must fight for access to the bus when there is only one bus, as there is in the von Neumann architecture. Creating a broader, quicker bus that can accommodate all requests without becoming a bottleneck is the conventional and accepted option (Figure 11.3).

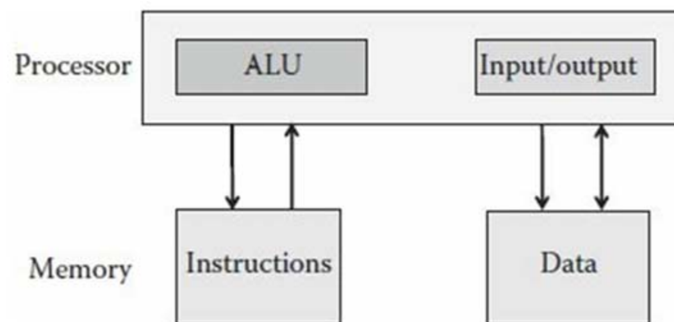

Figure 11.3: Harvard architecture.

The Harvard design, which makes use of a dual memory/bus notion and is discussed in Chapter 3's section on computer systems, was another suggestion for releasing the potential bus bottleneck (Figure 5.3). Using two distinct memories—one for instructions and one for data—enables the basic notion of removing uncertainties and reducing data-transfer bottlenecks. As each of these memories has its own bus, all possible conflicts in bus access are avoided. The other ALU—memory operations may continue while the control unit retrieves the next instruction and the operands without interfering with them.

Storage Sizes

Following Moore's law, memory sizes are continually growing and are utilised in contemporary computers as well as other digital devices like smartphones, cameras, smart TVs, and so forth. The early computers employed thousands of memory cells, even before the personal computer (PC) was invented. From the original PC's 640 KB memory use, memory capacity has rapidly increased. Both the applications and the tools they use have advanced, requiring bigger memories, while at the same time, memory costs are falling as Moore's law anticipated. Systems with ever expanding memory sizes are the end outcome. Although the higher-end sizes in the table are not relevant for memories (yet), larger memories are used by different virtual machines that are made up of many systems working in parallel, as is the

case with cloud computing, since the amount of memory currently in a single system is measured in gigabytes.

Memory management

Memory is, technically speaking, a digital storage system for data. According to this broad definition, memory devices include hard drives, CD-ROMs, disk-on-keys, and more. The focus of this chapter will be on the computer's random access memory (RAM), which, unlike the majority of other hardware, is primarily utilised for short-term data storage. All of the data that was saved in the RAM is lost after turning the system off and on again or just restarting the machine. Memory is hence referred to be volatile in contrast to other storage devices, which are referred to as nonvolatile, that continue to hold the data even when the power is switched off.

A one-dimensional matrix of cells that are utilised to store and retrieve data makes up the computer memory. Each cell has an address, which offers a special way to retrieve the information it holds. The address is used as the cell's index, much as in a matrix. The size of the memory cell may vary depending on which computer is used. It may be one byte (8 bits) at times or a word of different length at other times. When a memory is defined as being byte addressable, or when each address relates to one byte, the cell size is a byte. The least amount of information that can be obtained is likewise this (brought from the memory or sent to the memory). As the cell is an atomic unit, all memory accesses will be in multiples of 1 byte if its size is 1. All memory accesses, however, will be made in multiples of one word if its size is one word. The whole byte will be fetched from memory and the unimportant sections will be masked out when just a tiny fraction of it needs to be accessed. Think of a byte that has eight separate binary switches as an example. The whole byte will be loaded from memory and the CPU will apply an AND* instruction with the constant 0000 01002 when it is necessary to verify, for example, bit number five. If the outcome is zero, bit number five was zero or "False"; if it were "Yes," the result would be "True."

Memory was referred to as RAM in the early computer generations to distinguish it from serial devices like magnetic tapes. The name was selected to convey the significance of being able to access any cell immediately without having to read all preceding ones. Because of this direct access, it may be assumed that the access time is constant and independent of the location of the cell. Notwithstanding the fact that all memories provide immediate access, the term RAM is still in use today, decades after it was first created. A graphic illustration of the memory structure. On systems like PCs, where the memory may be accessed by a single byte, the registers are often bigger and based on words. Nonetheless, it should be emphasised that different systems utilise varying word sizes and that the term "word" has several meanings. It could be 16 bits in certain systems, 32 bits in others, like the x86-based architecture, and 64 bits in contemporary PCs. A bigger register suggests that it contains more bytes. The memory is remains byte accessible, nevertheless, regardless of the size or quantity of bytes in the register.

The memory address register (MAR), which stores the address of the bytes to be read from or written to the memory, is a crucial register for comprehending memory operations. This register is a 32-bit register, much as the other registers in x86-based computers. This indicates that 232 - 1 is the maximum address that may be used (or 4 GB of memory). It became clear in the 1990s that the word size would need to be extended to 64 bits owing to this restriction. While there were other systems that used 64-bit words, the PC industry, with its enormous variety of software packages, saw a far slower rate of development; the 64-bit versions didn't begin to surface until the first decade of the twenty-first century. The bigger word size offers a better level of precision when doing mathematical computations, "Data Representation."

Nonetheless, the memory restriction imposed by 32-bit systems was the primary factor in the move to a 64-bit architecture. It became clear throughout the 1990s that the PC platform could not advance without an increase in the maximum supported RAM.

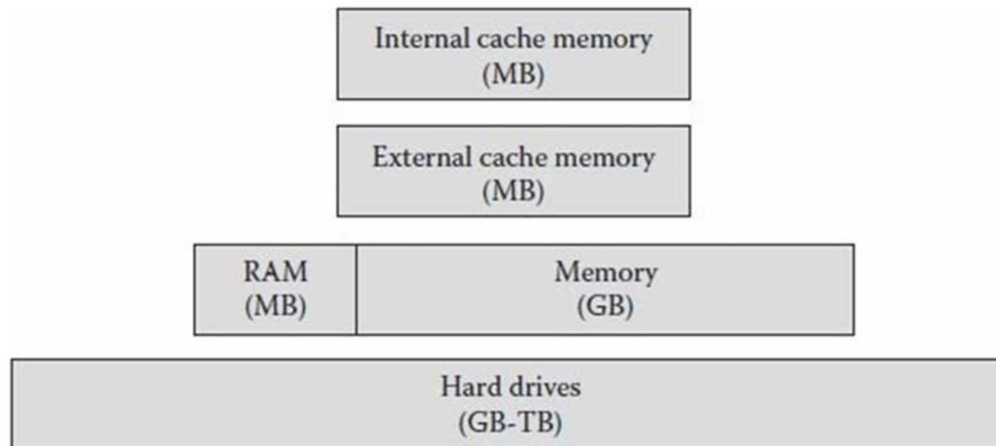

Figure 11.4: memory hierarchy.

There are many different forms of memory, such as volatile and nonvolatile memory, but a further significant difference is based on speed. The software developer chooses to purchase the most memory and the fastest memory accessible because they see memory as a component of the hardware platform. Of course, this is feasible, but it comes at a significant price. Because of this, the majority of computer systems use a hierarchy of memory. There is less memory near to the CPU since it is speedier and more expensive. It grows slower, more expensive, and bigger as it moves farther away from the CPU. In most cases, systems include a number of cache memories that speed up access to frequently used data and instructions in addition to the registers, which are a type of temporary, limited, and very fast memory that reside inside the processor. These cache memories will be discussed in more detail in later chapters. On the other hand, because only some of the running programme is in memory at any one moment and other, inactive portions are saved on disc, discs (hard drives, to be described) are sometimes seen as an extra sort of memory. Thus, the disc functions as an addition to the memory (Figure 11.4).

CHAPTER 12

DYNAMIC RAM (DRAM) TECHNOLOGY

Sowmya M S, Assistant Professor,
Department of Computer Science and Engineering,
Jain (Deemed to be University) Bangalore, Karnataka, India
Email Id- ms.sowmya@jainuniversity.ac.in

Dynamic RAM (DRAM) technology, which is defined by the necessity to refresh the data every few milliseconds, is often used to implement the main memory (RAM). For each bit, this sort of memory only needs one transistor and one capacitor. It is possible to create systems with billions of bytes since these components are so tiny and will only become smaller as technology advances. Although the cache memory does not need to be updated, it is constructed using a quicker technique (static RAM [SRAM]), which needs four to six transistors per bit. Because of this, cache memory is also more costly than regular memory.

With byte-addressable PCs, the memory access employs two internal registers, as shown in the following diagram:

Memory data register (MDR), used to contain the data to be put into the memory or the data read from memory, and memory address register (MAR), used to maintain the required memory location to read from or write into

The steps listed below may be used to describe a streamlined memory access. The processor must enter the value 1000 into the MAR and execute a read instruction if it wants to read data from address 1000.

The material will be retrieved by the memory controller and put in the MDR, where the processor may access it. But, if the processor has to store data in memory, for instance because of an instruction being carried out by the programme, it stores the data in the MDR, inserts the necessary address into the MAR, and then performs a write command. The actual execution of the necessary action is the responsibility of the memory controller. The read and write operations on the memory were originally designed for a single byte. Both the registers and the memory may be accessed by bytes.

The memory controller had to read or write 2 bytes at a time when the word size rose, for instance to 16 bits (or 2 bytes). Many hardware vendors accomplished this in various ways. For instance, the PC used a system in which the high-order byte was kept in the next byte place and the low-order byte was kept in the leftmost location. A 16-bit word is thus written into memory in reverse order. In the memory representation of the 16-bit word "AB," for instance, the first byte is the leftmost letter, while the second is the rightmost. This held consistent even as the word's size expanded, resulting in a 32-bit word containing "ABCD" being stored in memory as "DCBA" (Figure 12.1).

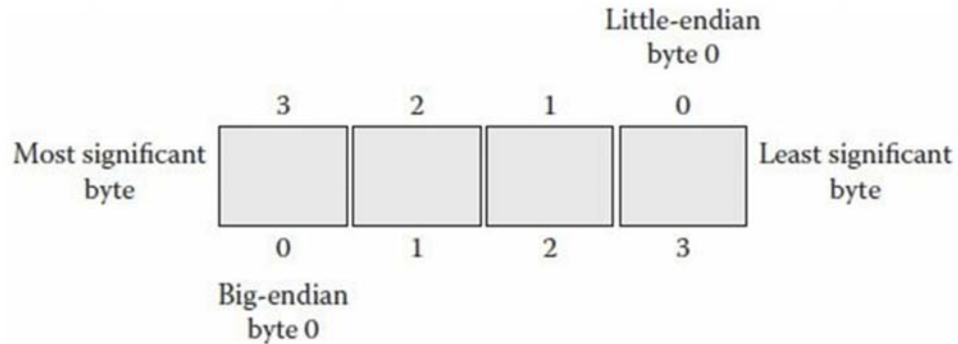

Figure 12.1: Big- and little-endian.

Although many other computers utilised the big-endian system, which is more logical and clear, this method, known as little-endian, was mostly employed by PCs.

Explanation

The 32-bit word is "standardly" written from left to right (big-endian), with the most significant byte on the left and the least significant byte on the right. The least significant byte is on the left when using the little-endian technique, while the most significant byte is on the right.

The big-endian technique is said to be more "natural" since it writes numbers from left to right, placing the most important digit on the left, and because when reading numbers, one often begins with the most important digit first. For instance, 57 is referred to as fifty-seven (it starts with the five). It should be noted, however, that the number is sometimes pronounced using the little-endian technique, beginning with the seven in the previous example, in certain languages (such as German, Danish, Dutch, etc.). Due to the compatibility of the PC architecture being preserved, the little-endian technique has survived. The memory controller initially had to send the 2 bytes one at a time when the word size was extended from 1 byte to 2 bytes. The little-endian technique was easier to implement and it is still in use, even if the memory controller presently only transmits one word at a time. Several suppliers adopted a dual technique due to the widespread usage of PCs and the fact that they use the little-endian method (bi-endian). This indicates that both big- and little-endian techniques were supported by the system, and that switching between them at boot time was conceivable. A software-based mechanism for toggling between the approaches was included in later versions.

It should be mentioned that there was another way that combined the two approaches in order to ensure the correctness of this description. A 32-bit word was used by the DEC-designed PDP computer. "BADC" was mentally recorded as a word that includes the letters "ABCD." This indicates that the word was split in two. The bytes in each half were written using little-endian, while the halves themselves were written in order (big-endian). Because of this, this technique was given the names middle-endian or mixed-endian. The majority of networks adopt the big-endian approach, despite the fact that PCs tend to favour the little-endian method. Since the hardware handles the data migration from one way to another automatically, systems and software developers don't have to worry about it (Figure 12.2).

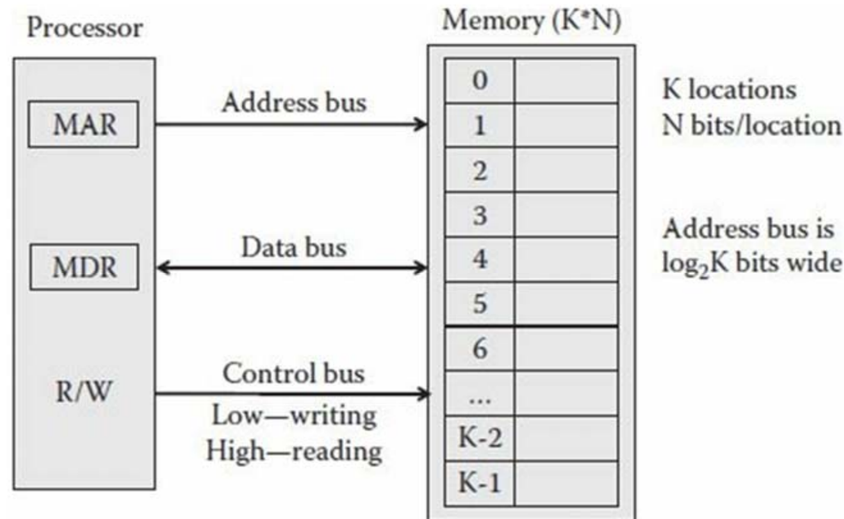

Figure 12.2: Memory buses.

The memory is arranged in bytes in a byte addressable architecture, and data is transferred to and from the memory in blocks made up of multiple bytes. Although it is feasible to read or write more than one byte in a single access cycle, it is not viable to do so. The software must read a byte and mask off any other bits that are irrelevant, as was previously stated, even if it only needs to access one bit. Buses serve as the foundation for the data transport system between the CPU and memory (this issue will be elaborated in the coming chapters). The bus is logically separated into three separate buses, each of which serves a different purpose. One is in charge of transmitting the appropriate address that was entered into the MAR. The data transmission is handled by the second bus. This bus has to be broad enough to fit the register's whole width. The third bus is used to indicate whether a piece of data needs to be read from or written to memory.

Each bus' width need to be enough for transporting the necessary data. Because of this, the address bus has to be large enough to accommodate the address of the last memory cell. In other words, the address bus' width should be Log2k if k is the final physical address. The data bus has to be big enough to suit the memory cell's size. It must be at least 1 byte wide in a byte-addressable architecture. The bus should be wider in situations when the registers are wider, such as 32-bit registers. If not, the bus will need to run four cycles in order to send the data into the register, which will impede both the transfer rate and the execution. Today's computers have data buses that are far wider than memory cells, allowing many bytes to be sent in a single cycle. The third bus may be somewhat small since it serves as a control signal (one bit to signal if it is a read or a write operation).

The internal registers that correspond to them ought to have the same size. Both the MDR and MAR will be the same width as their respective buses, with the MDR holding the data and the MAR holding the address. In comparison to the CPU, the memory and memory buses are much slower. The memory hierarchy and wider bus were both implemented for this purpose, among others. Nonetheless, new strategies were devised to avoid or reduce potential memory constraints. Through the use of several registers, which strengthen the memory hierarchy, the reduced instruction set computer (RISC) architecture was created with the goal of reducing memory access. Virtual operating systems add another software-based component to the issue, which will be described later in this chapter. Despite all the strategies used, memory access may sometimes considerably impair the performance of the systems.

Parallel systems with shared memory are one such example. Under these circumstances, a number of processors—sometimes a great number—are attempting to access the memory, and the available bandwidth may not be enough. In such circumstances, the bus width is occasionally extended in addition to the memory's extra buses. A system with two processors that use distinct sets of buses for every processor As a result, if one bus is busy sending data to one CPU, the other processor may continue using its own bus without having to wait. Due to the fact that there are two parallel-capable buses, this system is known as a dual-port memory. Similar to this, more buses may be added to link the CPUs and memory (Figure 12.3).

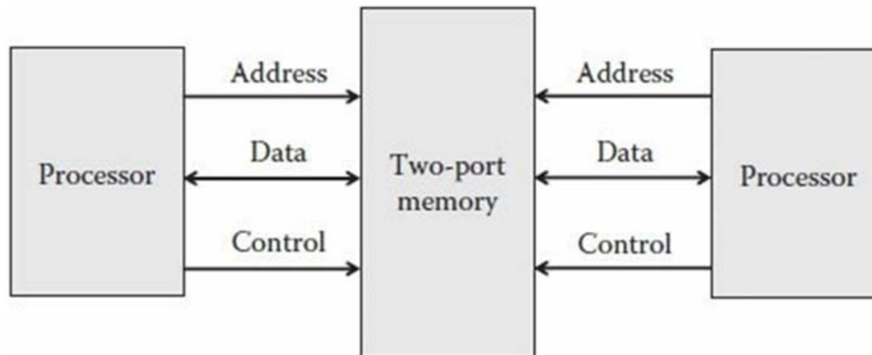

Figure 12.3: Illustrates the Dual-port memory.

Adding more ports (or buses) will incur extra costs, some of which might be substantial. The main cost factor is not related to the physical component of the system (the cable), but to the synchronisation mechanisms that must be implemented to prevent simultaneous writing from multiple processors to the same location or reading of data before it has been updated by a different processor (very similar to the pipeline hazards available in the CPU). a system having four processors, a shared memory, and several parallel buses. It was separated into four separate components, each of which was capable of operating concurrently with the others in order to increase memory speed.

Active Programs

The system execution and memory use of programmes have evolved throughout the years as a result of technical advancements. Just one application could be executed at a time on the early computers and the operating systems that supported them. The operating system was in charge of loading the application before execution. The program's whole memory set, as well as any system utilities it could have required, had to fit in the available physical memory. The software could not be loaded or executed if the memory space was insufficient. Due to the fact that only one programme was running at once, this form of operation was known as mono-programming. Due to the lack of tools to fully exploit the technology, this manner of operation was very ineffective.

It should be emphasized that at the time, computers were quite costly, therefore there waspressure from the economy to make the most of the system. If a computer system cost several million dollars and wasn't used to its full potential, a lot of money was being lost. Nevertheless, as the first computers could only run one application at once, this automatically led to an issue with utilisation. The processor was idle while the application was waiting for input or output since it was and is still far slower than the processor (Figure 10).

The mono-programming approach had the additional drawback that because the whole

programme had to be put into memory, the physical memory space needed to be sufficient to hold the biggest programme feasible portions of the RAM were left unused for several additional minor applications (Figure 12.4).

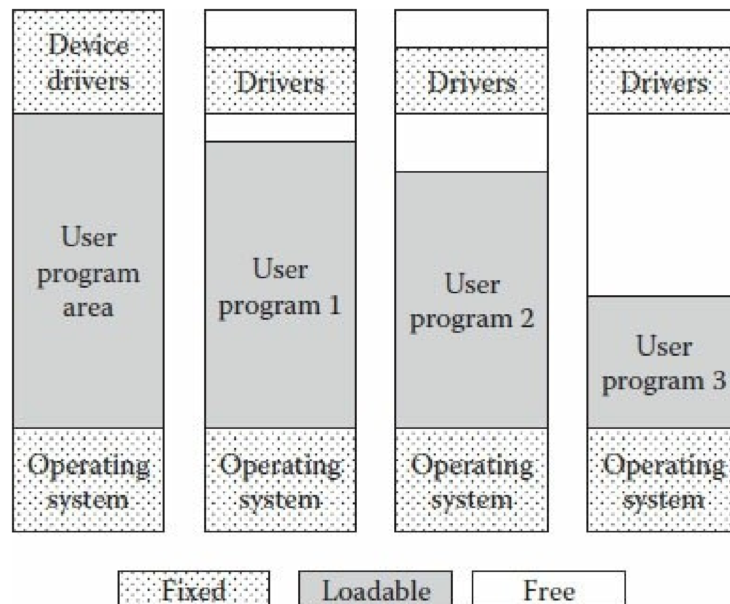

Figure 12.4: illustrates the Mono-programming: Memory content.

An immediate adjustment was required due to the expensive expenses of obtaining a computer system on the one hand, and the monoprogramming approach, which prevents the system from being used to its maximum potential, on the other. The enhancement allowed for the simultaneous operation of several programmes by modifying the operating system and making minor hardware adjustments. The switch from monoprogramming brought to better use of the expensive resources as well as the following additional advantages:

The programme will be split up into smaller programmes: It is preferable and more economical to execute numerous programmes sequentially rather than one lengthy programme.

The original computers had some reliability issues (hardware, operating system, and sometimes even application defects), thus if a problem sprang up during a lengthy programme run, it had to be restarted. With the development of interactive programmes and contemporary designs that utilise many processors and multiple cores, this shift in how people create their programmes took on significant importance. Moreover, as the field of software engineering has grown, once-massive applications have been replaced with a modular approach to simplify development and future maintenance.

Improved and more equitable resource management is possible using monoprogramming techniques, which allow one programme to execute at any one time while using all other computer resources. The operating system is in charge of creating methods for priority setting when several applications are running concurrently so that resources may be distributed across numerous processes with identical priorities. assistance with interactive programmes: Nevertheless, before it can be used, the system must be able to allow the execution of more than one application at once. Support for interactive use was eventually created. It should be emphasised that the word in parallel is inaccurate, and this will be clarified later. One programme was running at a time since there was only one processor available, but another

programme may continue while one was waiting for input or output. The interactive user believes that just their task is being carried out.

The concept of altering the system to accommodate many programmes comes from the knowledge that the computer is fast enough to handle numerous tasks simultaneously, allowing each activity to function independently and not be aware of the other processes. The phrase "time sharing" was created as one of the new features to describe how the system behaves when carrying out a number of activities. All active activities or processes are given equal time allocation by the CPU. The operating system determines how long the processor spends on each activity before moving on to the next, which receives its time, and so on. Many scheduling techniques have been created over the years to manage how much time each activity receives; however, as they are already included in the operating system, we won't examine them here. The operating system manages task scheduling, which includes judgements about which job will be done next, how long it will run for, how to assign priorities, and other things. In order to offer the necessary capabilities, additional hardware characteristics were needed, such as safeguarding task integrity and blocking situations in which a task attempted to access data that belonged to another job or the operating system. Also, the hardware needed to enable context switching, which entails preserving a task's whole execution environment as it waits to be executed. The state of the open files that are being utilised, the content of the stack, the contents of the hardware registers at the time of preemption, and other factors are all part of the execution environment.

The circumstance when just task A is running is shown in the diagram's top section. The bottom section shows the system during the execution of tasks A and B, while the centre part describes task B's execution. It is evident that the two duties are not carried out simultaneously (this is a single-processor system, so there are no resources to run two tasks in parallel). Only when the system is idle, or when the other process is not running, does one task execute. The fact that the processor's idle time reduced, though, is what matters. This implies that the system's productivity grew and its resource consumption increased, which improved the price/performance ratio. It should be noted that the second task (B) may not start immediately after task A has ended. It will have to wait for its turn in this situation, which could cause a little increase in reaction time, but the processor's idle time will still reduce. Due to the effectiveness of dual programming, it was only logical to improve the approach to support more jobs running "in parallel." The multiprogramming approach is this.

The processor's idle time continues to decrease, just as with dual programming. Nonetheless, it should be noticed that the illustration is really basic. In reality, nobody can guarantee that the tasks will be executed within the processor's free time. But, as was previously stated (in the case of dual programming), even if the jobs come at the same time, it is the operating system's obligation to handle them. Nonetheless, the higher workload will allow for improved system utilisation, which will enhance the price/performance ratio. The notion of virtual memory, which does not need the whole programme to exist in memory but just the executing portions, allows for a number of jobs to be completed simultaneously that is substantially more than three on the majority of contemporary computers.

Increasing the number of jobs running simultaneously is possible as long as the resources are sufficient. The basic assumption for adding more tasks derives from numerous insights: Most applications need more resources since they don't only utilise the CPU. Often, particularly with interactive systems, input data is required, and output is produced. Moreover, the software may need data saved on the disc during operation (hard drive). As compared to processing speed, these gadgets are very sluggish. It may take a very long time when humans are involved in delivering the input; in this case, the software is waiting for input while the

user is on the phone or has already gone for lunch. If the system adopts a monoprogramming strategy in any of these scenarios, system utilisation will be very low. The first computers were pricey and little memory. While this is not a problem in current computers since memory capacity has improved significantly and its cost has decreased significantly (one of the effects of Moore's law), we nevertheless appreciate the advancements that were necessary to solve earlier issues. This is hardly the only instance in which new technology was needed to solve an issue that no longer exists.

Since the CPU and the whole computer system were quite costly, much focus was placed on making the greatest and most effective use of them. Due to the affordability of modern computers, this issue is now something of the past. Also, the price/performance dilemma is no longer relevant due to the PC revolution. Many purchase PCs because of their inexpensive cost and only use them sometimes. Overuse was replaced with quicker response times and improved user experience. Another historical problem was the realisation that switching between activities might sometimes be time-consuming and impractical. Since then, the hardware has changed, more context-switching instructions have been introduced, and task switching overhead has decreased. Yet, each of these context switches still incurs some cost that must be taken into account. For instance, the overhead is higher in current architectures with many of registers since the system, among other things, has to save the contents of these registers and then reload them with fresh information.

Calculating the Usage of the Processor

The proportion of time that tasks or processes spend waiting for input or output must be estimated in order to determine the processor's utilisation under a given load. While determining the precise needed architecture, a system engineer may find it useful to consider the processor's usage. Nevertheless, if the system is to be utilised by users and there are hundreds of such systems to be installed, more dependable decision making is needed. On the one hand, the falling cost of hardware makes it feasible to decide on a much bigger configuration.

The estimate is given by the formula,

$$\mathbf{CPU_Utilization = 1 - p^n}$$

Where,

P is the average percentage of time processes are waiting for I/O n is the number of processes executing in the system

More specifically, the detailed formula is,

$$\mathbf{CPU_Utilization = 1 - \left(p_1^* \, p_2^* \, p_3^* \cdots p_n\right)}$$

Where

p1 is the percentage of time the first process is waiting for I/O

p2 is the percentage of time the second process is waiting for I/O, and so on n is the number of processes

For example, assuming

p = 0.8 (on average the processes are waiting for I/O 80% of the time) n = 4 (there are four processes executing) the processor’s utilization will be

$$\text{CPU_Utilization} = 1 - 0.8^4$$

When deciding whether to upgrade a system, using the CPU utilisation formula may be a useful tool, providing that the memory capacity of the existing system restricts CPU consumption. This occurs when only a small number of jobs can fit in the free and available memory capacity. Calculating the impact of this update is straightforward since the cost of the RAM increase is known. The system is on the left shortly after the boot procedure is finished. The system has 1 GB of memory, half of which is utilised by the operating system and the other half is free for usage by user programmes.

The same system is shown in the diagram to the right, but in this instance, there are four user processes in the memory. For the sake of this illustration, we'll suppose that, given the average size of each process, the memory's vacant space can hold four processes. The following potential diagram shows an improved machine with twice the amount of RAM. This system has 2 GB instead of the standard 1 GB. Instead of the four processes in the 1 GB architecture, there are now nine user processes in this system due to an increase in the number of continuously running processes. The original system is shown in the final (rightmost) after having undergone two upgrades; the memory capacity has been increased from 1 GB to 3 GB. Once again, the number of user processes rises, and now the system can execute 14 processes simultaneously rather than the four in the original design and the nine in the 2 GB system. The scale on the right is not related to system cost, but rather to memory size and expenditures specific to memory.

The processor will use 59% of its initial configuration if each of these processes waits for I/O on average 80% of the time. The number of processors and the usage of each CPU increase when the system is upgraded by the addition of 1 GB of RAM. At n = 9, the processor's usage rises to 87%, as shown by the formula. It is easy to determine if the increased performance achieved by raising CPU usage by 28% (87% - 59% = 28%) can be justified by the higher cost since the upgrade cost is known. The third architecture is subject to the same examination. There are 14 concurrent processes when the memory capacity is increased to 3 GB. The processor's utilisation will rise to 96%, and as before, the upgrade's value can be assessed based on the system's improved performance since the upgrade cost is known. It may be advisable to utilise a 2 GB system in this particular scenario since the first update results in a 28% performance gain whereas the second upgrade only results in a 9% boost in performance.

The key problem with processor utilisation is that all of these calculations and evaluations can be done without a real upgrade, making it possible for the system engineer to find the ideal solution without shelling out money for a real upgrade or using any of the benchmarks that are currently available. All of these estimates, however, are predicated on the notion that the site's operational behaviour is understood. In reality, estimating the amount of time a process spends waiting for I/O is really challenging. Moreover, the aforementioned approach disregards the viewpoint of the user. A CPU usage of 96% may provide better price/performance if the mentioned system is employed for interactive jobs, but the reaction time will probably be rather long. Given the time users spent waiting for the system to respond, the money saved by increased usage may not be beneficial in this situation.

Partitions

Multiprogramming may be implemented via a number of ways. Creating and using divisions is the first and simplest strategy. A partition is a section of memory that has been set aside for storing running programmes. It is a specific kind of operating system. The partitions were first set during the boot procedure. The size of each partition was unique and could only be

modified at the time of the subsequent boot. The amount of free RAM on the system for user processes is equal to the sum of the partition sizes. The number of processes that could run in a given partition was determined by its size. Just some of the divisions were usable at the time since the whole process had to be loaded before the virtual memory idea was invented. The number of processes that may be run simultaneously depended on the number of partitions that were established. The operating system was in charge of managing the partitions and selecting which software would be loaded to which partition. The partition concept sought to emulate the original monoprogramming computers by dividing the computer into a number of specified working contexts, each of which was meant to resemble a monoprogramming system. The entire amount of free RAM was split by the number of partitions in the initial partition implementations, which employed fixed-size partitions. The creators didn't realise the drawbacks of monoprogramming with fixed- and identical-size divisions until much later. The next step was to specify divisions with different sizes. The partitions continued to imitate the monoprogramming approach by using this notion, but with several enhancements. The operating system was responsible for managing the partitions and making decisions depending on

Which partition offered the greatest match based on the processes' memory needs. The operating system needed to be informed of the number of partitions, their sizes, and the size of each process that would be run in order to apply these modifications. Each process or application that needed to run had to declare how much memory it needed, and the operating system utilised this parameter for allocating resources. ,

The system's memory properties are shown in the diagram on the left. It is a 1 GB system in this instance, and the graphic shows the free memory that may be used by user programmes. Device drivers, which should have been loaded at higher locations in memory, were left out of the diagram to make it simpler. to the right after the open memory space has been partitioned into four unique divisions of varying sizes. As mentioned before, with fixed partitions, increasing the number of partitions or their sizes requires a fresh system boot. A list (queue) of priority processes (identified as 1-7) that are awaiting execution is shown at the top of the diagram. The scenario is shown in the following after the operating system has begun assigning partitions to the running and awaiting processes. Although though it can be observed that, in most circumstances, the process's memory needs are smaller than the partition size, which results in unused memory, the operating system assigns each of the first four to the partition that fits the situation the best.

The scenario is shown in the following to be later. After Process 4 has finished running, the operating system must allocate the next process. The scheduler assigns the huge partition to the little job, squandering a significant portion of the partition's memory, even though Process 5 only needs a small amount of memory and there is only one partition available. However, in this example, the scheduler decisions are based on only one parameter, the next process in line (this is an example of a simple first come first served [FCFS] algorithm; see the footnote in the section "Running Programs" of this chapter). It should be noted that modern operating systems may have elevated wisdom regarding the scheduling process, for example, making decisions based on some additional parameters.

While it wasn't ideal, the fixed-partitions approach gave us the ability to run many processes simultaneously. The dynamic partitions, a more complex process, were made possible thanks in large part to its most significant contribution, which was to prepare the way. The realisation that fixed partitions continue to squander memory resources drove further advancements. With one advancement over fixed partitions, dynamic partitions are based on

them. The idea behind dynamic partitions is that the operating system may modify the number and size of partitions while it is running normally. The goal of this was to increase the number of continuously running processes in order to improve memory management and CPU usage. The rise in operating system overhead necessitates the creation of a new method for managing "holes," which are portions of unallocated memory that must be maintained, relocated, and merged. This is only one of the unfortunate drawbacks.

The system's memory properties are shown in the diagram on the left. There is a 1 GB system once again, and the graphic shows the free memory that is accessible to user programmes. The system after the open memory space was partitioned into four divisions of varying sizes is shown in the following (to the right). The list of processes that are awaiting execution is shown at the top of the diagram. The scenario after the operating system has designated a partition for each process is shown in the following diagram. In this instance, four processes are loaded into memory since there are only four partitions. The operating system notices that some of the partitions have gaps at the ends after assigning the four processes. Moreover, Process 5 (the procedure after it) takes up relatively little memory, which may be gained by merging the remaining holes. The two top partitions are moved by the operating system in such a manner that every hole is joined to form a single continuous memory segment. The operating system allots Process 5 a new partition that will be utilised for the new memory segments. The operating system creates a new partition and loads Process 6 into it since there is enough RAM left over to load Process 6 as well after all six processes have loaded. Processes 4 and 6 eventually complete running, at which point the RAM set aside for them is designated as free. The operating systems use several strategies in relation to holes that are being generated. The fundamental guideline is to avoid doing excess labour that is not required. For of this, the operating system seldom employs dynamic holes management. The operating system may shift the partitions to make a bigger continuous section, according to the dynamic holes management. The typical approach is hole management by demand, where the operating system will only shift the partitions to make space for a bigger piece of memory if a task specifically requests it. The handling of memory gaps in this particular case is done on demand. It should be noted that holes management adds overhead to the system, therefore for dynamic holes management, the operating system raises overhead, but it is prepared for increased memory demands in the future. The overhead is reduced with holes management per demand, but it will take longer to process requests as they come in.

Afterwards, Process 1 makes a request for more RAM. Certain new features were implemented as part of the operating system updates to assist the user in better managing the process' memory needs. Memory allocation (MALLOC) is one of these processes. Regrettably, Process 1 is utilising its whole size inside of a partition. It's also possible that the operating system reallocated the unused RAM that was assigned to a separate partition after the process was moved to a bigger partition. The operating system may roll out Process 1 up until a bigger partition becomes available in order to provide the required RAM. Process 1 will be rolled back in once the RAM is available so it can start operating once again. In the scenario shown, the operating system determines whether it can accommodate the request and discovers that partition 3 leaves a sizable gap. The operating system then raises partitions 2 and 3, produces the newly required piece, and combines partition 1 with it.

CHAPTER 13

ELECTRONIC MEMORY

Krishnan Batri, Deputy Director,
Department of Computer Science and Engineering,
Jain (Deemed to be University) Bangalore, Karnataka, India
Email Id- krishnan.batri@jainuniversity.ac.in

The strategies discussed in the preceding sections were designed to make it possible to perform many jobs at once. Even bigger memory could only hold a certain amount of jobs as the size of the programmes grew, which created additional issues. The primary driver of technical advancements, as previously said, was economic. The earliest computers were quite expensive, therefore the computer centre manager had to ensure that the system was used as much as possible in order to justify the cost. In an attempt to provide a better price/performance ratio, computer companies created the multiprogramming techniques. Their attempts were impeded by the then-current technology, which relied on loading the whole programme into memory. While the memory size could be extended, there was a large cost involved. Also, it was discovered after closely examining the behaviour of the programme that was now running that every programme includes substantial components that are seldom or never utilised. For instance, certain portions of the programme are used for managing errors, and it's possible that during a particular run, no errors need to be handled. Because of this behaviour, it was realised that it was not required to load whole programmes into memory since alternative techniques might be designed to load the missing pieces as needed. Another

The growth of the market for off-the-shelf applications was a key incentive. Even if a system's physical memory capacity is constrained, these programmes should be able to operate on it. Under these circumstances, it could run more slowly, but it should still be able to move. The ability to execute a programme that requires more memory than the physical memory available is made possible through a number of ways. Allowing the coders to handle the problem was one straightforward option. As the developer is already acquainted with the software and is aware of its execution patterns, a customised solution may be created. However, such solutions may have system-specific restrictions. The usage of an overlay mechanism is an example of such a strategy. The programme may be broken into many pieces (overlays), and each of these portions can be performed separately from the others. This is the basic premise. Instead, the programme may be created such that it employs many components that are performed one at a time. When utilising a machine with a limited amount of memory, only one portion will be loaded at a time, and after it has finished running, it will be emptied to make room for the next part to be loaded and performed. The overlay process is still utilised by various gadgets and mobile phones even though current computers have adopted other, more advanced and effective techniques. Assume, for instance, that a particular programme is broken up into six components, each of which is in charge of a separate action. The first three parts operate separately; that is, part one never interacts with parts two or three while part two is running or does not refer to data that is present in parts one and three while part three is operating. Although sections five and six are similar, they are autonomous. On the other hand, component four serves as a common foundation for the whole programme and offers the services needed by all the other parts. The

traditional and straightforward technique included loading every component into memory, which is a wasteful action. A more effective strategy would be to provide three overlays. Parts one, two, or three would be loaded using the initial overlay. The common infrastructure (part 4) would be loaded using the second overlay, and parts 5 or 6 would be loaded using the third overlay.

We'll use an actual case to provide further context. assuming that 63 KB of RAM is needed to execute the application on an embedded system. This embedded device can hold just 32 KB of memory due to several limitations. Overlays were used to close the gap between the physical constraints and the programme memory needs. The system uses 15 KB worth of common tables and variables. Since this common component must be accessible to all other components, it will be loaded for the duration. The first section, which uses 5 KB to initialise the system, is only used at the start of the run.

The second section needs 17 KB, the third section needs 16 KB, and the fourth section needs 10 KB.

Although working independently, parts two, three, and four need access to the shared components. This indicates that the components don't communicate with one another or access information kept in a different portion.

The available 32 KB of memory were split into two parts to reduce the amount of memory needed. The first is a 15 KB static chunk that is used for common data. A second 17 KB chunk will also be utilised for the code. Of course, the second chunk has to be big enough to fit the biggest code overlay (the second part in this example). The first half is loaded into the second chunk during startup. Based on how the programme behaves, the same chunk will be utilised to load the remaining code components when initialization is complete.

It should be emphasised that the overlay method can only be used when the programme can be split up into separate executing pieces and there is total separation between these parts. In order to establish the necessary separation, it is sometimes essential to duplicate comparable functions and methods. Of course, this duplication violates one of the fundamental and crucial rules of good programming methods. The industry decided to search for an alternative strategy as a result of the redundant code duplication and the realisation that manual optimization was not the best course of action.

Nowadays, a method known as virtual memory is used to automatically execute a programme even when there is not enough physical memory. The underlying and established presumption is that programmes include a large number of pieces for handling a wide range of scenarios and instances, but that only a tiny portion of the code is actually used while the programme is running. For instance, a banking application that manages checking accounts does not need to execute the code that addresses all potential incorrect scenarios if all withdrawals are legal throughout programme operation (invalid date, insufficient funds in the account, etc.). The fact that certain particular components of the programme do not need to be in memory frees up memory for other, more beneficial uses.

Creating virtual memory necessitates creating a system that loads just a portion of the currently running apps while loading the other portion as needed. Of course, this applies if the system can quickly load the missing components. Since the sections that are not needed will stay on the discs (hard drives), they are considered to be a component of the memory hierarchy. While a programme is running and the system freezes, we may view some of the application's components that are in memory while other components are still stored on the disc.

Because the way in which the memory is organised differs greatly from how we often think

of memory, the technique is known as virtual memory. The memory is a one-dimensional matrix of sequential cells, as was previously mentioned (see the section "Memory Organization" in this chapter). Yet, the utilised cells in the virtual memory are not arranged in a linear fashion. It takes a unique hardware technique to convert virtual memory addresses to actual addresses. While physically these cells may be located in the memory in various and presumably not continuous places, the programme illustrates the developers' perspective of the system in which the memory is a continuous array of cells. For each memory access during execution, the translation between the virtual and physical locations is done instantly.

Below are the definitions of the virtual and physical addresses for a more thorough explanation:

The software creates a virtual address during execution by accessing a variable, executing a function, or using a method. The address of the method or variables becomes a part of the instruction after compilation. As it's conceivable that the necessary material is not in memory since it hasn't yet been loaded, this address may be regarded as a virtual address. The location of a particular cell or procedure in memory is indicated by its physical address. The hardware must convert the virtual address to a physical address whenever the software attempts to access a particular virtual place while it is being executed. The hardware also determines if the necessary address is present in memory as part of the translation. In this situation, the access will be made. If the address is not in memory, the hardware will interrupt, which the operating system will handle. The necessary material must be loaded by the operating system. The programme is suspended during the loading process so that other applications may access the CPU. The halted application won't be allowed to continue on running until the data in the address has been loaded. The overlays notion forms the foundation of the virtual memory technique, which contains a substantial enhancement. The two processes vary significantly in two ways. With virtual memory, the software is separated into fixed-size components, and the hardware and operating system handle these components automatically. Let's say, for instance, that an application needs 5 MB of RAM. Pages are fixed-length segments inside the software. In this example, 64 KB pages will be used. This indicates that there are 80 pages in the programme. Not every one of these pages has to be in memory for execution, as was previously indicated. Moreover, the physical memory is split up into 64 KB-long units called frames. Such frames are designed to hold one sheet each. The designers are attempting to strike a compromise between two opposing trends while deciding on the page/frame size. On the one hand, memory is used more effectively the lower the page/frame size (less unneeded parts that are loaded). On the other hand, loading pages with a tiny page/frame size necessitates additional input and output procedures. Also, the operating system keeps a database with the addresses of the pages, and as memory sizes grow, this table expands significantly. A big page table adds to the overhead of the system and reduces the amount of memory that can be used by user applications. Because of this, several manufacturers are thinking of increasing the page/frame size, which would cut the size of this table in half. Several systems supported various page sizes in the past, when computers were substantially more costly and systems were in use for a longer period of time. This indicates that the page size was preset at boot time and could only be altered at the next boot. Nowadays, the page/frame size is fixed, and the majority of systems no longer provide this functionality. In most cases, the page size is a power of two.

The preceding example is shown in Figure 20; the programme and its pages are on the left side, and a system with less memory is on the right. Although not all pages are loaded into the physical memory, as shown in the picture, the addresses used by the programme are virtual addresses. Also, the loaded pages of the programme are spread out throughout the physical memory. We will suppose that the software performs the instructions on page zero

when it first starts up. This indicates that the first frame will be loaded from page zero. The page was loaded into frame zero since it was the first frame that was accessible in this particular situation. The programme keeps running and eventually reaches or invokes a function that is located on page 4. The hardware can't finish the address translation since this page wasn't in memory, so it generates an interrupt. The programme is placed on pause by the operating system when it recognises the sort of interruption. The missing page is then located and loaded into the next open frame. The page is loaded into frame one in this particular instance. The software won't be able to continue running until the page has loaded. As in this case, the programme keeps running and eventually reads page zero once again. The software proceeds without being aware that it is operating on a different place in memory since the page is in memory and the hardware has already translated the virtual address to the actual address. The programmes eventually reach page six. The page is not in memory once again, therefore the hardware alerts the operating system to load it, and so on until the programme is complete (Figure 13.1).

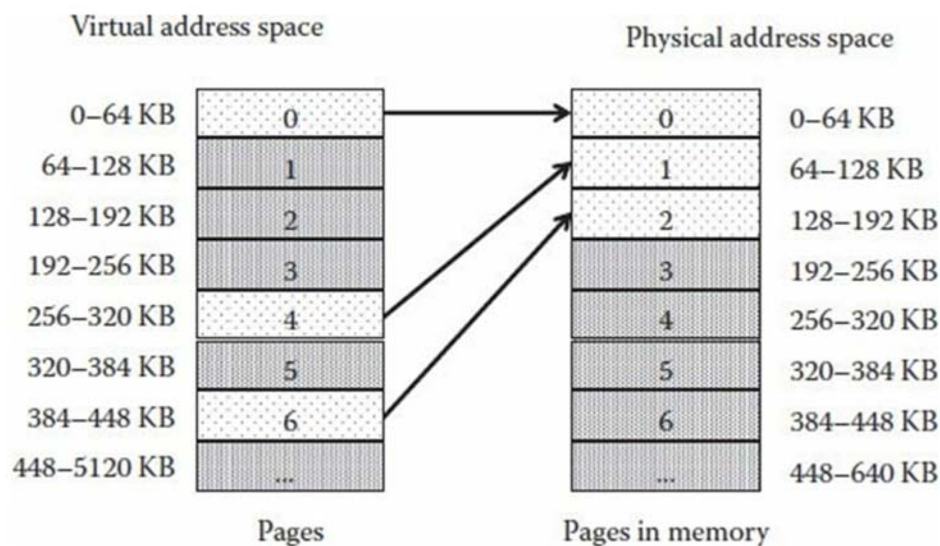

Figure 13.1 Pages and frames example.

The operating system that loads the necessary pages manages every frame in the system and employs a number of techniques to optimise the execution of applications. For instance, it is in charge of avoiding scenarios in which one application completely controls the system and consumes all memory. In other situations, the operating system may preempt frames from another application that is executing on the system if frames are not immediately accessible. The operating system uses a variety of techniques to manage the frames, but they are beyond the scope of this book. Yet there's also another problem that systems engineers should be concerned about. The generated programme should be modular and separated into functions, classes, and methods as part of excellent software development practise. Each method or function should be quite simple, since this lowers complexity and makes development and maintenance easier. Another justification for keeping the functions and methods somewhat compact comes from understanding the virtual memory mechanism. Smaller code chunks increase the likelihood that the whole function or method will be included on a single page, resulting in fewer pages being loaded during execution.

The MMU is in charge of managing memory connectivity and converting the virtual addresses found in instructions into actual locations in the memory. The CPU is running a

programme that requires access to a virtual address. The address might include a variable required by the now running instruction or the next instruction. A continuous array of cells with virtual addresses is how the software conceptualises its address space. The virtual address is sent to the MMU by the CU, which is in charge of retrieving both the next instruction and the necessary operands. To determine its true location, it is translated there. The MMU also determines if the page is in memory as part of the translation process. The necessary cell will be moved from the memory to the CU, if it is in memory (Figure 13.2).

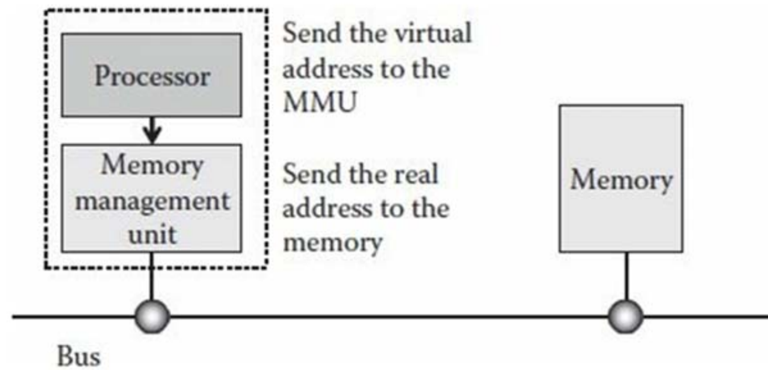

Figure 13.2: Address translation.

We shall use a condensed illustration based on a theoretical system to more clearly grasp the translation process. The greatest address that this theoretical system may allow is 31 if it uses 8-byte pages and each application is limited to using no more than four pages (addresses starting from zero to 31). This indicates that this system's address space is based on 5 bits. There will be two groups made up of these five parts. As the programme can only utilise a maximum of four pages, the first group will be used to define the page number, using two bits in total. The second component will be used to specify the page displacement. The address space will need 3 bits since the page size is 8 bytes. The system for catalogue numbers used in a warehouse may serve as a relatively straightforward real-world example. Let's suppose that each element has a distinct five-digit number that may be thought of as a concatenation of two values. The component's location is indicated by one number (two digits), while the appropriate shelf is indicated by the second value (three digits). It is pretty obvious where the exact component is by looking at the catalogue number. Although the programme addresses are sequential and the virtual address space is consecutive, the address may still be divided into two integers. a fictitious application of 13 bytes. The application requires the addresses 00000-01100. The programme bytes are described on the left side of the diagram. The address is represented by the first two columns on the left, and each byte of data is represented by a distinct character in the right column. The two leftmost columns are joined to form the virtual address. For instance, the letter "a" is located at address 00000, whereas the letter "k" is located at location 01010.

BUS

This chapter concentrates on and goes into detail about many features of buses, which serve as the system's backbone for data transport. We can understand how the system's buses relate to the overall architectural figure. The method for tying together different functional units in the system is known as a bus, sometimes referred to as a channel. Several of the connecting and data-transferring components have previously been covered, such as the processor, which requires memory data for both its instructions and operands. Another such example is the multiple memory types that need to be linked in order to transfer data effortlessly (outside of

the control of the operating system or the application that is now executing). Many buses are used to fulfil the von Neumann architecture's emphasis on modularity's requirement for a data transmission method.

The bus is a network of electrical cables that transport data in parallel. Such wires convey a single bit. Depending on its width, the grouping of wires may transport a word or a byte at a time. The bus adheres to a set of protocols or rules that specify ownership, priority, who may transfer data and in what sequence, and other aspects of its operation.

It shows five separate functional units (devices) connected by two different kinds of buses, which are each represented by an ellipse. Each device in the network of connections shown in the diagram on the right is directly linked to every other device. As one transfer between two devices does not affect the transmission of data between other devices, this kind offers extremely excellent transfer speeds. One bus links all of the gadgets on the left side. In this situation, every linked device must use the same protocol. The single bus solution is often less expensive, slower, and demands more administration than a network of connections. Each device that requires data transmission must first get access to the bus before acting in accordance with the set regulations. Each device must request permission before using the bus since it is a shared bus that can only transport one block of data at a time. A railway system is a practical illustration of how to operate a bus. Assume that there is just one railroad connecting two sites and that trains go both ways. There is only one railroad in the area, thus before departing from the station, each train must confirm there are no other trains utilising the line. Any device that needs to utilise the bus must follow the same rules. The device must first request authorization to utilise the bus; only after receiving authorization can it begin data transmission. Actually, the transfer itself is utilised for sending orders (read, write, etc.) transferring the device's opposite end's address (just like calling a number when using the telephone system) actually moving the data (after communication has been established) A bus may either employ the distributed arbitration shown in Figure 13.3 or management from a single point (central arbitration). One central unit (the arbitrator), which controls the bus transfers, is what defines central arbitration. Any device that wishes to use the bus must first request permission from the arbitrator, and only after receiving that authorization may the device utilise the bus for its intended uses (see Figure 13.4 and 13.5). The bus is logically separated into three distinct channels since there are many different sorts of transfers that might occur on it:

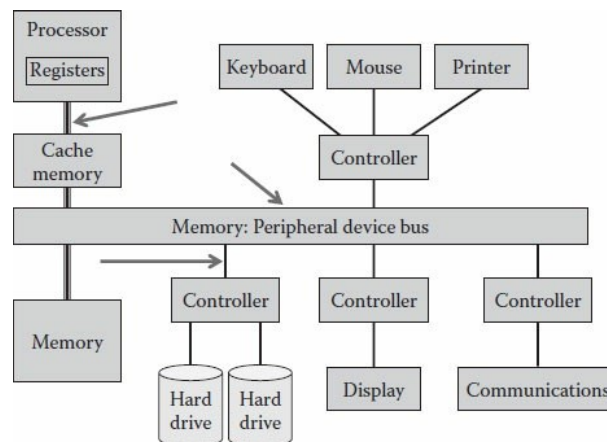

Figure 13.3: Buses as part of the architecture.

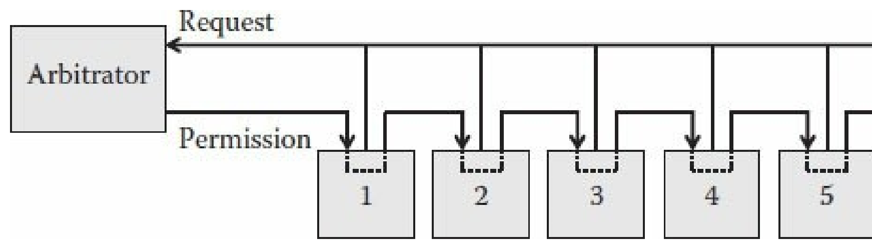

Figure 13.4 Central arbitration.

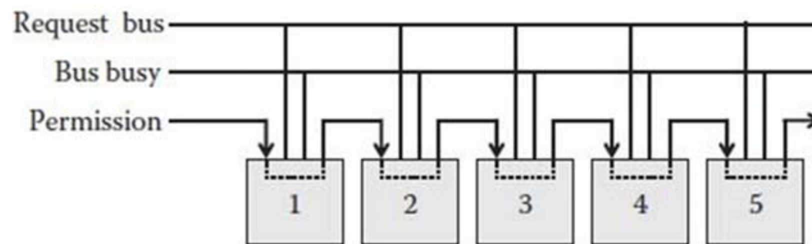

Figure 13.5: Distributed arbitration.

One is for control, and the apparatus utilises it to transmit orders, the third is used for data transfers, the second is utilised for addresses.

Each device is linked in serial and is listening to the address channel. The gadget responds when its unique address is broadcast. If it is not their number, the other devices keep listening but disregard it. It is comparable to mobile phones in that they listen to the network and only answer when their number is called.

The absence of centralised control over the activity of the bus characterises distributed arbitration. Instead of using a centralised unit, all of the devices work together to manage the system while adhering to the established protocol. The device must still request authorization to utilise the bus in a distributed arbitration, but the confirmation procedure is somewhat different. There are three channels that make up the bus, plus an extra one for indicating whether the bus is in use or available. Each device monitors the bus's free/busy signal and waits till it changes. The request is not sent till after that. The requests are received by the other devices on the bus, and they respond in accordance with a set priority scheme. The device is in charge of switching the bus signal back to free when the transfer is complete so that other devices may utilise it. In actuality, there are other algorithms for guaranteeing seamless communications, including priority management and hunger prevention, therefore it is crucial to keep in mind that this is just a very basic explanation of the priority algorithms.

A device may utilise one of two techniques for sending data: Complete transaction In other words, the device requests the bus, and once it receives it, it retains it during the whole connection. This implies that a number of blocks may be transmitted back and forth during this period. The responding device responds after the asking device transmits the address. The other device responds to the request in the next step, and this process continues until all necessary data has been received. The bus will only be released by the asking device after all communication has been completed. Similar to line

communication, this style of operation keeps the line open for the length of the conversation even when nothing is being exchanged across it. Due to the crowded nature of the bus during this transaction, only extremely quick devices and brief communications may use this sort of communication; all other devices that need it must wait.

Split transaction: This describes how the device requests the bus, then, after receiving permission, sends a command, an address, or data before immediately releasing the bus. In situations when there is a delay between the processes, split transactions are desirable. Consider the scenario when a block of data from the main memory is required by the cache memory. Cache memory controller will request authorization before using the bus. After the request is approved, the cache controller will request the block from the main memory. The cache memory controller will release the bus after this instruction transmission. The main memory controller requires time to process the request, thus this is done. It's conceivable that it has a backlog of pending requests that must be handled before the one in progress can be handled. It will take some time for the main memory controller to access memory and acquire the necessary block, even if the present request is the first in line. The main memory controller won't request permission to utilise the bus until the data is ready, and once that request is approved, it will transfer the block to the cache memory controller. This manner of operation is comparable to contemporary cellular networks, which share the channel among several phones. Each pair of phones creates a virtual channel utilising the bus (assuming there are no conference calls). Such virtual channels may operate simultaneously on this bus in a time-sharing manner. The key concept behind the split-transaction approach is that the virtual channel will release the bus so that someone else may use it if there is nothing to transmit.

Depending on the kind of bus, there are two methods to transfer data across it: A clock-synchronized synchronous bus is one example. All bus-connected devices utilise a continuous signal that is present on the control channel. All activity on the bus, such transferring data, is timed to the start of the signal. A command, address, or data transmission will take place for one or two bus cycles (Figure 13.6). The synchronisation signal is described in the first (upper) line, and the subsequent lines show that all other operations (read, confirmation, write, etc.) are synchronised and begin at the start of a cycle.

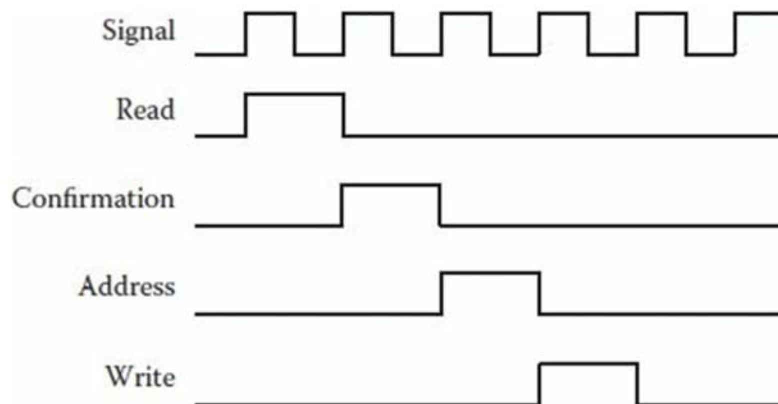

Figure 13.6: Synchronous bus.

An asynchronous bus is the alternative kind; it is not synchronised, and each action comes after the one before it. The rate is determined by the sending device. This method is more

difficult on the one hand, but it also enables the connecting of numerous devices that operate at different speeds.

Transport Policy

A computer called a programmed data processor (PDP), created and manufactured by Digital Equipment Corporation, included the bus for the first time in 1964. (DEC). Large, costly computers that needed specialised cooling systems dominated the computing industry at the time. The PDP, subsequently referred to as a minicomputer, was very distinct. In contrast to mainframes, it was inexpensive (costing tens of thousands of dollars compared with hundreds of thousands and even millions). It had integrated a number of ground-breaking concepts, such as the ability to function in a typical air-conditioned environment without the need for additional cooling equipment (Figure 13.7). A communal bus was yet another concept put into action (called omnibus). The computer systems utilised channels before the bus was invented, which established a matrix of connections where each device was linked to every other equipment via a dedicated cable (quite similar to the figure on the right). Cost might be reduced by having one bus that all devices could utilise, but this also created additional bottleneck issues, which will be covered in the next sections.

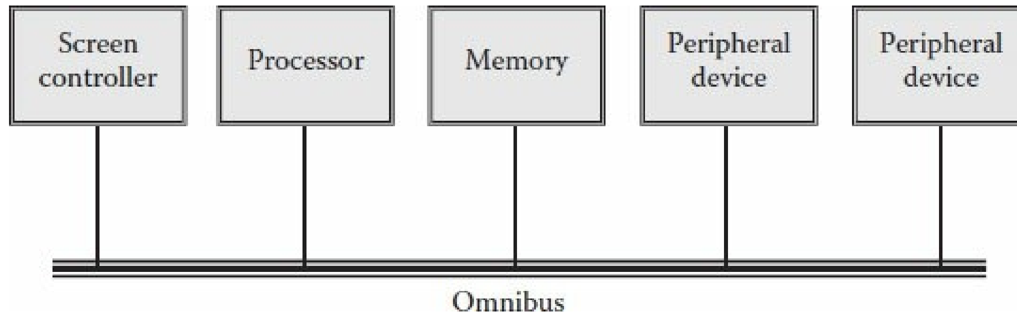

Figure 13.7: depicts the PDP architecture with its common bus.

It should be emphasised that the bus physically consists of a collection of wires that preserves this logical separation. The bus concept had a big and crucial role in the growth of the computer sector. Before the personal computer (PC), a few of businesses who produced and marketed turnkey systems dominated the computing industry. Since the utilised buses and channels were proprietary, other businesses were unable to supply any tools or parts to link to these private systems. PCs were created utilising standard buses, which enable a limitless number of components and devices that may be smoothly attached to the system, in contrast to this method. Considering that their solution is superior, many suppliers, even tiny ones, have access to the market in the hitherto closed environment. Due to the intense competition there, as we all know, prices were able to be lowered in this open market.

A computer made by any vendor may be connected to memory made by other vendors, discs made by yet other suppliers, and so on (Figure 13.8).

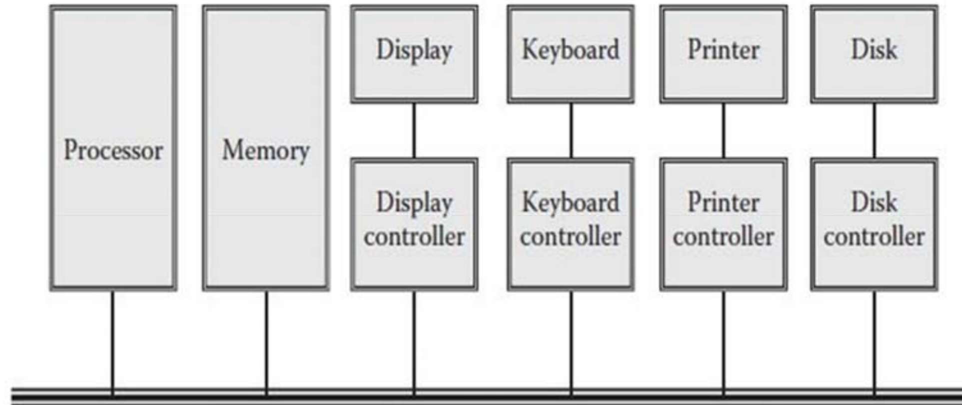

Figure 13.8: Architecture of the first PCs.

Similar to how the PDP's standard bus was first established, the PC's standard bus was initially designed to reduce the PC's cost. In fact, the original PCs had an architecture that is similar to the PDP architecture. The architecture's controllers were primarily used as a bridge between the devices and the bus protocol (keyboard, display, disk, etc.). As each device has an own operating system, it is necessary to translate and integrate the devices so they can operate in a bus environment. Unless the device is a standard device, which may already have an off-the-shelf controller, the controller is typically developed by the device maker.

Together with the financial benefits of employing a bus, this method also makes it simple to swap out devices and move them from one system to another (assuming the two systems use the same bus). However, as technology improved, certain benefits turned into drawbacks. The bus became a bottleneck as additional devices, particularly those with different speeds and transfer rates, were attached to it. For instance, there may be instances when the bus is occupied transmitting a character entered on the keyboard while the CPU waits for data to be transmitted from memory. Similar to memory, a hierarchy of buses was established depending on their needed functionality as the chosen approach:

A processor-memory bus that must have the highest transfer rate achievable This is often a private bus set up by the motherboard* manufacturer. The bus that connects the processor to the memory can become a serious bottleneck, which will have a negative effect on performance (for example, by increasing the CPI; see section "'Iron Law' of Processor Performance" in Chapter 4). This has already been seen in previous chapters on memory and cache memory. The bus speed and bandwidth are crucial because of this. The length of the processor-memory bus is often quite short owing to the near proximity of the CPU and memory. This small distance is another element that improves performance. The distance the electrons are travelling begins to matter for time at the speeds of today.

an input-output (I/O) bus, which is often a readily available bus. By doing this, the number of I/O devices that can be attached to the system is increased. Since some of these devices are physically huge and cannot be placed near together, this bus is often longer than the processor-memory bus in order to give the room for connecting the different components. The I/O bus further supports a wide range of devices in addition to devices that operate at various speeds. The speed variations may vary by many orders of magnitude in certain circumstances. In most cases, the I/O bus is connected to the

processor-memory bus or, if one is present, to the system bus. The I/O bus is implemented in contemporary systems as a hierarchy of buses (as will be explained later in this chapter).

On the motherboard is a system bus, which may be either standard or proprietary. In certain configurations, the I/O bus and processor-memory bus are coupled instead of the system bus. The other two buses, however, will be attached to the system bus if it is there. The system bus may sometimes be designed as a hierarchy of buses, much as the I/O bus.

CHAPTER 14

BUS DEVELOPMENT

N Sengottaiyan, Deputy Director,
Department of Computer Science and Engineering,
Jain (Deemed to be University) Bangalore, Karnataka, India
Email Id- sengottaiyan.n@jainuniversity.ac.in

As previously mentioned, the initial bus implementations (in the PDP in the 1960s and in PCs in the early 1980s) used a single bus, to which all of the system's devices were linked. Using a single bus has clear benefits (simplicity, which led to a lower cost). The bus, which controls all data transfers throughout the system (from the CPU, memory, discs, keyboard, etc.), became a bottleneck. The bus was significantly slowed down by the several different devices attached to it, and the CPU was also adversely affected. The length of a single bus influences its time. Even when a bus operates well, the amount of data sent by all the various devices is already near to the bus's full capacity. The speeds of the different devices also cover a wide range of values. For instance, a keyboard that relies on human clicks can only generate around 10 keystrokes per second. An eight-bit character, or a maximum bandwidth of 80 bits per second, is translated into each of these keystrokes. Tens of billions of bits per second are required by the CPU that operates at a 3 GHz clock rate, which is on the upper end. There will be an I/O bus and a fast processor-memory bus in a system with two buses, and they will both be linked via a specialised adapter to bridge the two (Figure 14.1). A system with several buses will always employ a hierarchy between the buses. The fast bus, for instance, links the CPU and memory in Figure 14.2. The processor-memory bus will be connected to other buses. The adapter, which functions as a controller of some kind, synchronises and connects to the two buses and their protocols. Typically, a device like this has a number of internal buffers that are utilised to temporarily gather data from the slower bus. The adapter won't deliver the data as a single block across the faster bus until the buffer is filled. In this manner, the processor-memory bus was not interrupted until the entire block was prepared for transfer, despite the fact that the data was gathered from slow devices. Despite the fact that this mechanism lessens the effect of the slower devices on the processor-memory bus, some systems use additional buses to further lessen any potential harm.

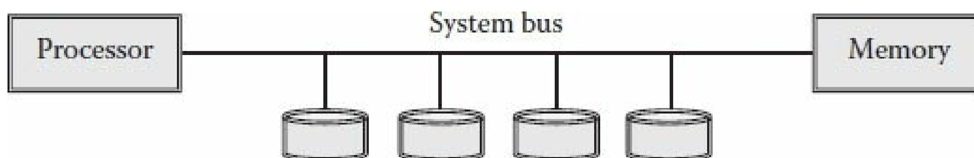

Figure 14.1: A system with a single bus.

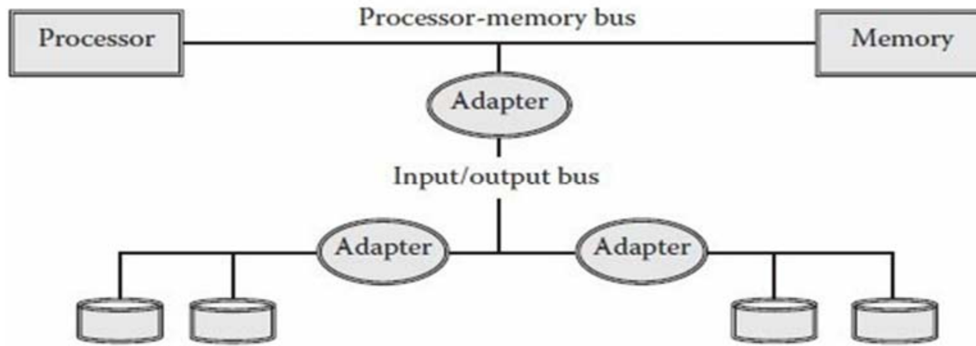

Figure 14.2: A system with two buses.

In the system shown in Figure 14.2, which has three buses, the processor-memory bus is only utilised for communications between the CPU and the memory. While the processor-memory bus is linked to the system bus, this connection is only utilised when a device needs to transfer an input block into memory or when the processor needs to access an input or output device. The system bus, which connects the other buses, is the primary bus in such a system. Depending on the linked devices and their characteristics, the system bus may be connected to a number of slower I/O buses. I/O buses are subject to a variety of standards. An open architecture results from the standard's ability to link a range of standard devices that adhere to the same standard. This kind of design provides the framework for combining diverse components made by several vendors. The majority of contemporary systems offer multiple buses in addition to extra interfaces that enable the extension of these buses. More standard buses may be connected using the extra interfaces or adapters. These adapters do protocol conversion. According to Figure 14.3, the adapter connects to one bus on the one hand and an extension bus on the other. As the extension bus is a regular bus, it may connect to an extra adapter that is attached to another bus since the extension bus is a regular bus. To link a wide range of I/O devices with the greatest amount of flexibility, a bus hierarchy may be constructed.

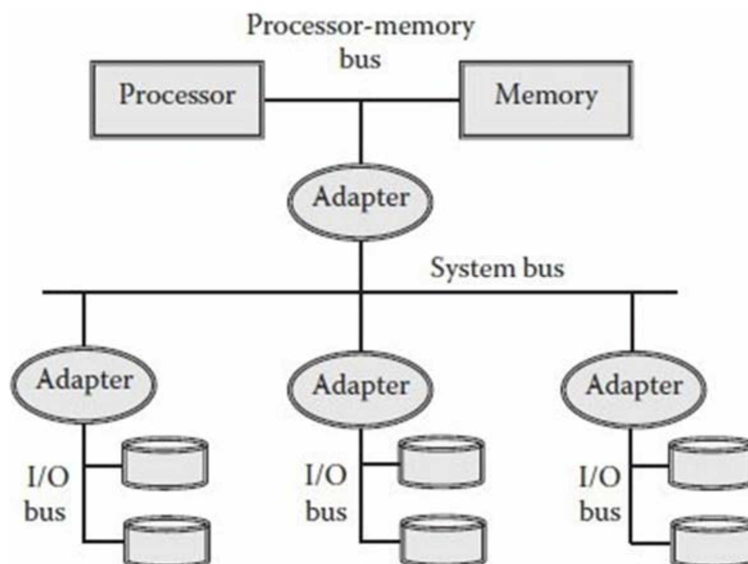

Figure 14.3: A system with three buses.

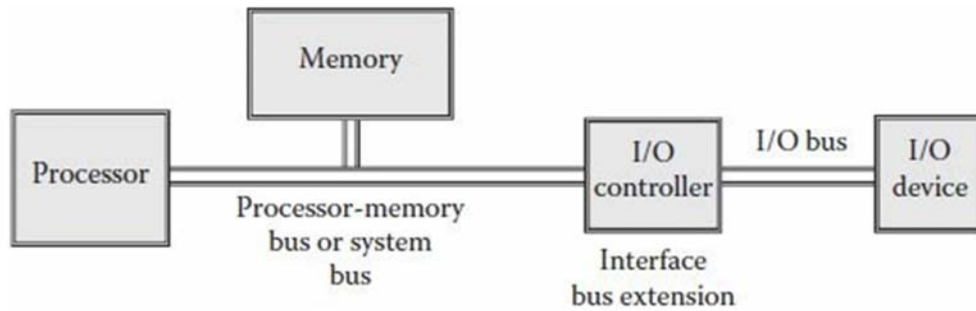

Figure 14.4: An extension bus.

In the example shown in Figure 14.4, the I/O controller performs the function of a protocol converter. The particular I/O device is on one end, and the bus to which the adapter is linked is on the other. The controller, as will be further discussed, not only controls and keeps track of the I/O device, but also gives the device the capacity to be connected to a number of common buses (at least the one the specific controller supports). In order to make the device run on a range of buses, the controller is often created and supplied by the device maker. In certain circumstances, such as with hard drives, the controller is an essential component of the disc and is sold together with the disc.

Most PCs include certain motherboard-based buses, and to connect the controller, one must place it in one of these slots, as seen in Figure 14.5.

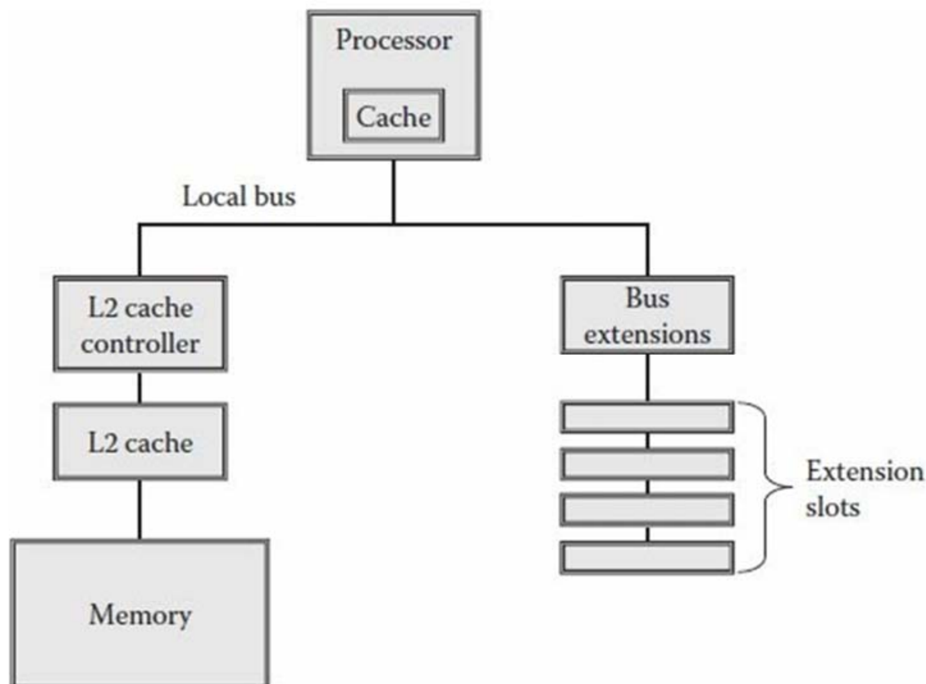

Figure 14.5: Extension slots.

Standard buses come in a wide range, and there are more and more of them. Industry

Standard Architecture (ISA), Extended ISA (EISA), Micro Channel Architecture (MCA), Personal Computer Interconnect (PCI), Single Inline Memory Module (SIMM), Dual Inline Memory Module (DIMM), Accelerated Graphics Port (AGP), and Personal Computer Memory Card International Association are a few of these standard buses and protocols (PCMCIA)

U.S. Serial Bus (USB)

These standards are only a few instances of the many solutions developed, and how these solutions are constantly changing as technology advances. The hierarchy of the buses is supported by the range of standards in addition to the increased flexibility in connecting the many I/O peripheral devices. This indicates that the link offers a productive approach to use the swift buses at their maximum speed with very little disruption brought on by the slower vehicles. The original plan was to simultaneously provide a common infrastructure for data transmission across all system components and serve the processor's data requirements without any delays. The ability to repurpose outdated gadgets and their controllers also has a positive economic impact. For instance, there has been almost no modification to keyboards throughout time. There is no need to create new keyboards that can function with the new buses since a suitable adaptor may be used instead (Figure 14.6).

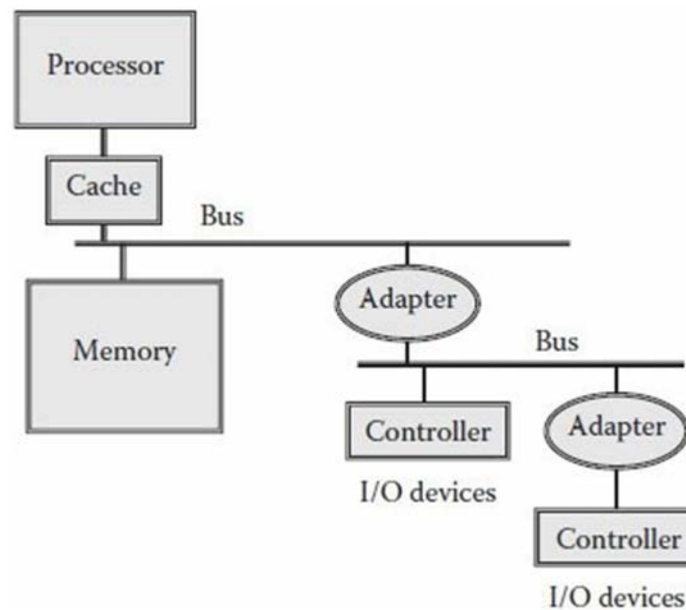

Figure 14.6: Bus hierarchy.

A system that supports a hierarchy of buses (Figure 14.6). The CPU and its cache memory are located in the top left corner. As there are no extra devices linked to this bus, there is a direct connection between the two (instead of a conventional bus). Typically, this connection will be lightning-fast. A fast bus connects the cache memory to the memory (such as the processor- memory bus). The main memory, the cache memory, and an adapter that links the lower-level buses are the only things connected to this bus. This bus will be used to transfer data from the memory to the cache memory and the processor whenever the data needed by the processor is not present in the cache memory. The adapter occasionally uses the bus to transfer data, but this only happens on rare occasions, and when it does, the adapter operates at bus speed. The I/O bus, which links many controllers, is located on the bottom bus. For example, the slow communications controller that connects to the bus and, on the other end,

may connect to a variety of devices or even additional communication buses, these controllers connect the peripheral devices or they can be used to bridge to further lower and slower buses (these are not part of the figure).

To reduce data transfer delays to the processor as much as possible was the primary need that sparked the development of the bus hierarchy. The memory architecture and hierarchy were designed with this as a key consideration. In order for data from a slower device to reach the processor, the bus hierarchy must, on the one hand, allow open access. On the other hand, the slower devices must never cause the processor to sluggish. Figure 14.7 serves as an illustration of a less-than-ideal design.

The processor and cache memory are connected by a very fast bus (a backdoor channel set aside only for communication between the two), but the processor is connected to the north bus. The north bus is also connected to the processor-memory bus. In this way, the north bus functions as a system bus and receives the majority of the system's data transfers. This configuration has a problem because the processor and cache memory cannot connect directly and must instead go through the north bus, which is shared by slower devices. In order to provide the necessary fast communication, the cache memory also needs to be directly connected to the memory, which is absent in this example.

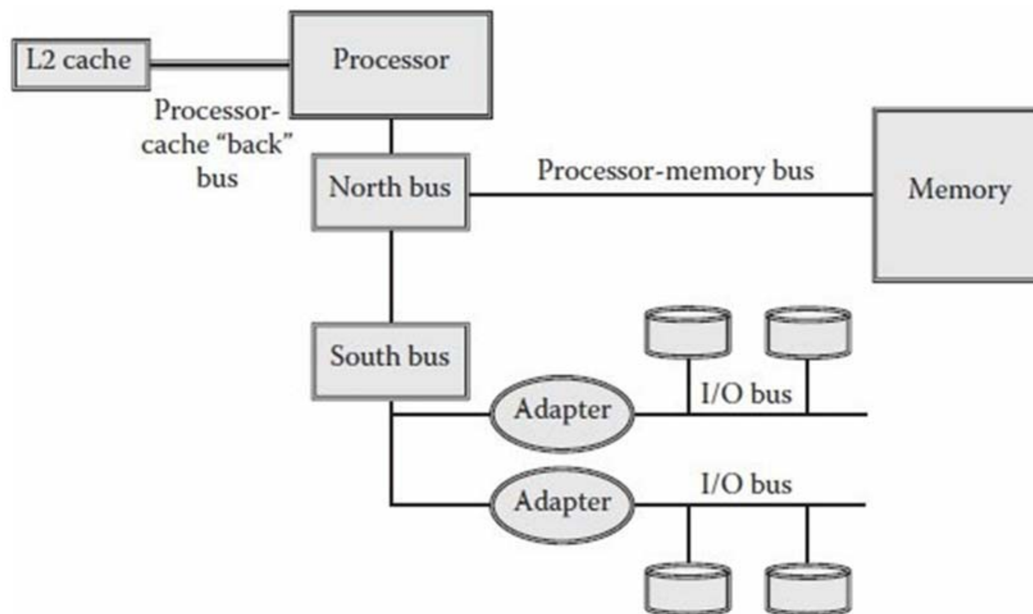

Figure 14.7: Bus conflicts.

One of the Pentium-based systems' bus design is seen in Figure 14.8. The processor's Level 1 on-chip cache may be found in the right top corner. The two caches are connected via a specialised rapid communication route (Level 1 and Level 2). This component has a quick 64-bit system bus connecting it to the CPU and the two caches. The Level 3 cache, the graphics adapter, and the main memory are all connected via this bus. This bus has a graphics adaptor to handle extremely quick visual displays, such as those used by gamers. Ordinary monitors will be linked via a slower USB if the system does not need these very quick displays. As was previously mentioned in this chapter, the bus speed and width have a significant impact on the memory speed. The width determines the unit of data, while the speed determines how many units of data may be sent every cycle (byte, word, etc.). In the

case of an 800 MHz bus, for instance, this would imply that 800 million blocks may be transferred across it per second. These blocks are each 64 bits wide. Wider buses are used by designers to achieve higher transfer speeds even though they are more costly (they need more wires in the cable and a bigger connection that takes up more space on the motherboard)

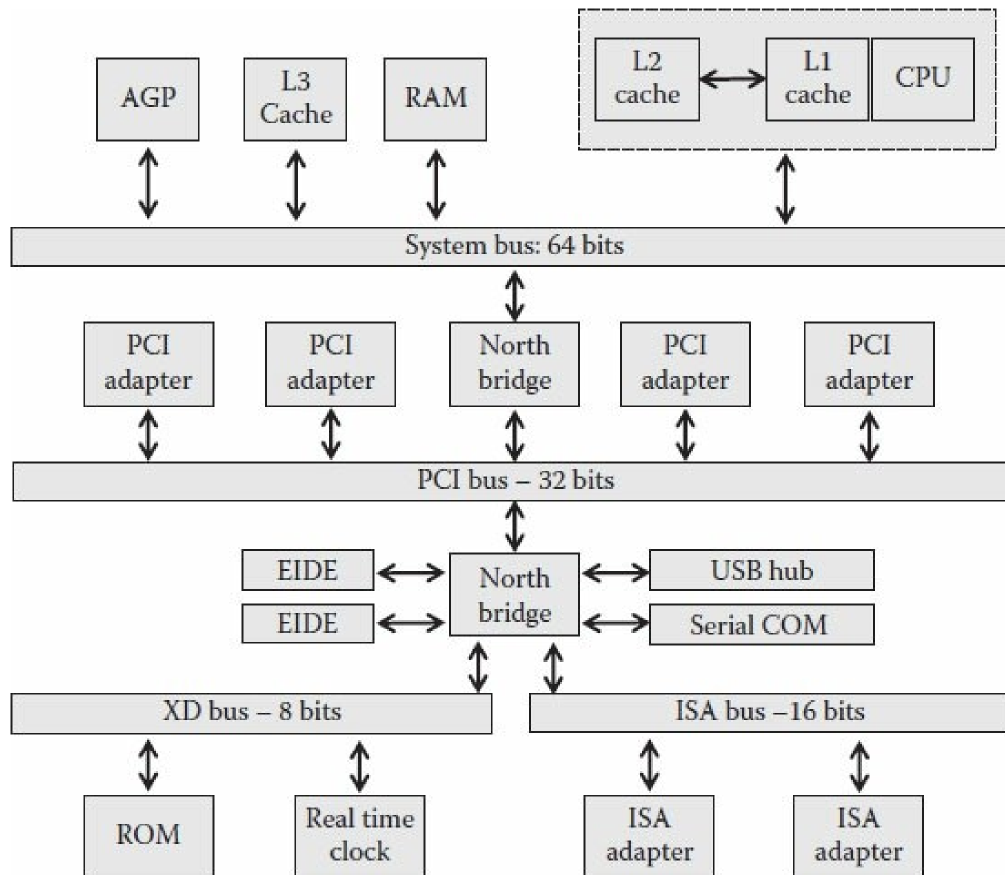

Figure 14.8: Pentium architecture.

A 32-bit PCI bus is linked to the fast system bus (through the north bridge). The north bridge connects the two buses, modifies block sizes and speeds, and translates protocols. A total of four extra adapters are included on the PCI bus, which may be used to connect PCI devices or bridges that enable the connection of slower buses. The PCI bus is linked to the south bridge on its lower side, which also links other, slower buses. Backward compatibility is one of the key characteristics and a key factor in the success of the PC series of computers. Several antiquated and even obsolete peripherals, including keyboards and floppy drives, are nonetheless supported by modern computers. Because of this, the south bridge, which links to current buses like USB on one side and the PCI bus on the other, also offers backward compatibility by connecting to older dated 16-bit and even 8-bit buses. This enables the reuse of certain peripherals that, although operating slowly, do not need replacement. One example of this is a real-time clock, which synchronises system processes while only transferring a little amount of data over the bus.

The timing characteristics of the buses near the CPU are further described in Figure 14.9. One cycle is needed to access the internal (Level 1) cache, which is the same as accessing registers. Depending on the technology and proximity, it may take two or three cycles to

bring the data to the second-level cache. If the information is not present at Levels 1 or 2, it must be retrieved from memory. In this instance, depending on the implementation details, bus conflicts, and other factors, the "cost" might range from 50 to 200 cycles (up to two orders of magnitude). The illustration highlights the value of cache memory once again for system speed.

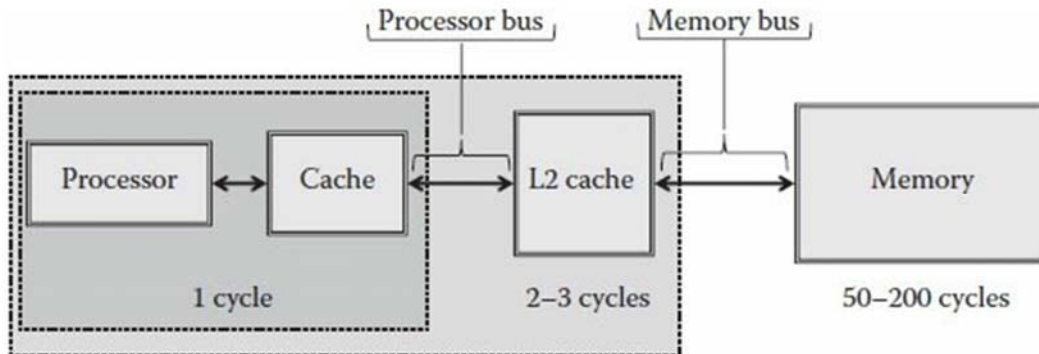

Figure 14.9: Faster bus access time.

Coaches for hard drives

Disks, particularly hard drives, optical discs, solid-state discs, and disks-on-key are some of the most significant technologies to have emerged concurrently with CPUs and memory. Numerous manufacturers focus on discs and provide a wide range of components that may be incorporated into many systems (computers, handheld appliances, mobile phones, cameras, etc.). Several standards for linking explicitly discs were created to give the necessary connection. Dedicated buses were required because the discs' distinct and sometimes differing characteristics. Buses made exclusively for discs were produced over time in an evolutionary method that closely followed technical advancements, much like regular buses. Some of the most significant advances are very new due to the recent quick progress. At Attachment (ATA), Integrated Drive Electronics (IDE), Extended IDE (EIDE), Small Computer Systems Interface are only a few of the previous and present disc transports (SCSI)

Due to some of its inherent shortcomings, several of these standards are no longer in use. The standard buses must be improved in order to support the new features and capabilities due to the steadily increasing demand for faster and higher capacity discs. It should be emphasised that the wide range of buses is a reflection of the severe rivalry that exists between different manufacturers, as each one strives to "push" their goods and expand market acceptability. Hard-drive manufacturers often use this to market their goods rather than solely to create new standards. The computer industry favours open standards in the twenty-first century because they enable sharing of devices made by many vendors. Even when a custom product is better, the consumer often chooses a standard product. SCSI is one of the buses that was often used for high-performance computers and is now mostly utilised for servers. The market position its creators aimed to occupy is reflected in the name of the product. IBM has made extensive use of this bus, which was designed many years ago, in an effort to distinguish its products and provide better performance. A parallel channel is used to join and concatenate several devices (Figure 14.10).

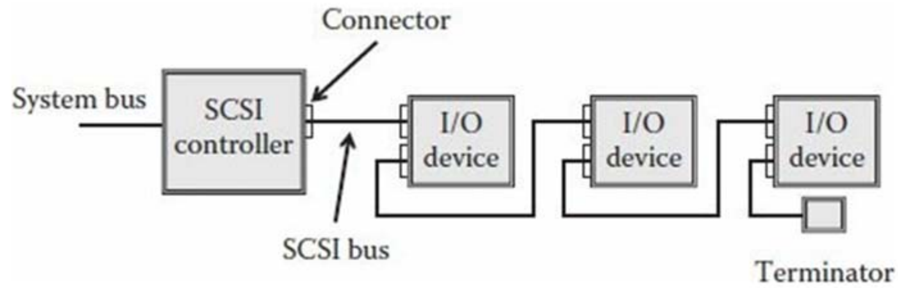

Figure 14.10: The SCSI bus.

Generally speaking, the bus is no different from other buses. It has an adapter (SCSI controller) that bridges and transforms the buses (such as PCMCIA or USB and SCSI). Concatenating several gadgets on the bus is one of its benefits. The last component is linked to a terminator that marks the bus's termination (Figure 7.19, right bottom corner). The bus's relative high throughput compared to other available buses contributed to its success. The standard saw a number of advancements, and over time, several versions were created, much like other devices and buses on the PC market. The majority of the time, the modifications were made to accommodate features like quicker transfer rates, a higher maximum number of devices per bus, etc. It was done to link a range of other devices as well during the process of designing and constructing specific buses for connecting mass storage (hard drives). The most prevalent PC buses, interfaces, and protocols are In order to illustrate the quick changes related not only with the processor's or memory technologies but also with all devices linked to the system as well as their connection infrastructure, it was intended to present a partial list of technologies that were and are currently in use. With the very basic serial bus, a more thorough description of the connections and protocol is given.

CHAPTER 15

CLASSIFICATION OF PARALLEL BUS

Merin Thomas, Associate Professor,
Department of Computer Science and Engineering,
Jain (Deemed to be University) Bangalore, Karnataka, India
Email Id- merin.thomas@jainuniversity.ac.in

Parallel Bus

Early PC machines included the serial bus. The serial bus operates on the principle of bit-by-bit data transmission. If a 1-byte datum has to be sent, one bit will be sent once per bus cycle. Eight bus cycles are required to send the whole byte. Although though this method of communication is obviously quite sluggish, it is straightforward and effectively explains how the Computer communicates. One of the communication (COM) ports was used for PC connectivity. The peripheral devices were attached to four of these ports on the first computers. The serial ports' communication protocol was known as RS-232. The Electronic Industries Association (EIA) created it to encourage using and connecting electronic devices. Using a modem, which acts as an adaptor between the computer and the telephone network, was one early example. A distant piece of data communication equipment (DCE) must be connected to the data terminal equipment via a modem and a phone line (DTE). As illustrated in Figure 15.1, the DCE stands in for the computer and the DTE for the terminal. The major goal was to build the initial networks, which opened the door for the present offers and can now be done extremely simply and at considerably greater speeds (Figure 15.2).

.

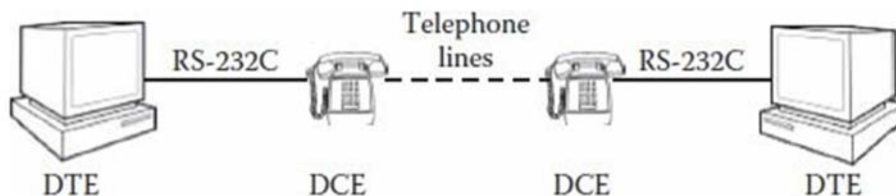

Figure 15.1: Serial communication.

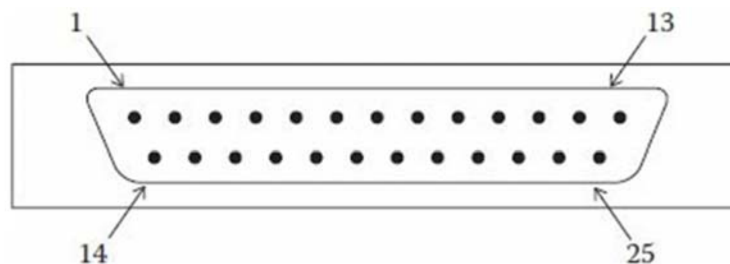

Figure 15.2: RS-232C connector.

Like any other bus, the serial bus may operate synchronously, in which case an external signal synchronises the transfer, or asynchronously, in which case bits are sent sequentially at

the bus speed. RS-232C* protocol-compatible connection was utilised to connect the devices (Figure 15.3). It should be noted that the PC's serial bus operates in full duplex, which allows data to be sent and received at the same time. Two different lines are utilised to send and receive data in order to permit parallel communication. One pair of wires is all that is required when the bus is operating in half-duplex mode, which allows it to transmit or receive data. Due to the telephone connections' limitations (a twisted pair of wires), the early PCs could only communicate in half-duplex. Just bits 2 and 7 are used for half duplex. Computers may be connected to the serial bus, which was created to link numerous devices and set the stage for telephone network connection. It made it possible to join two computers in an easy (rudimentary) method without using a phone connection. Of course, the closeness was quite little, and it shrunk much more as the transmission speeds rose.

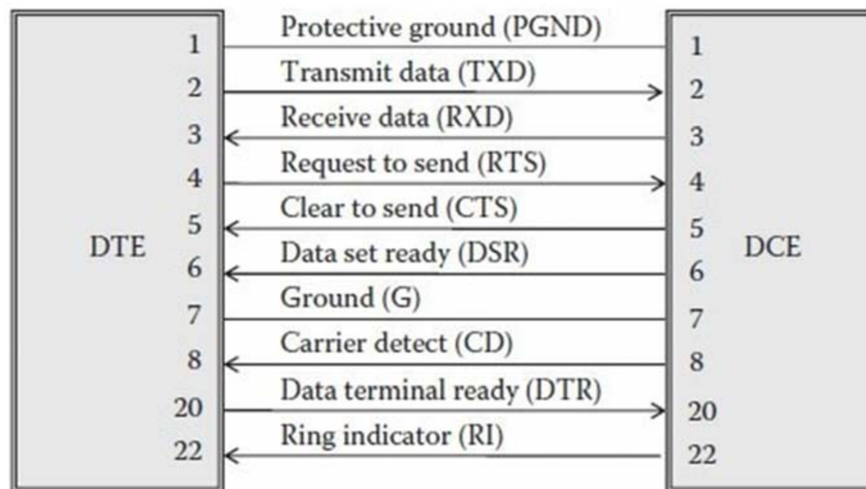

Figure 15.3: RS-232 connection meaning.

The following is the meaning of the different bits (signals) transmitted on the bus:

A reference signal called protective ground (PGND) is utilised to synchronise the voltage on the two sides. To establish if the other signals are turned on or off, they are compared to this reference. The PGND is linked to the ground in certain connections (G). The signal known as "transmit data" (TXD) is used to send bits of data from the DTE (computer) to the device. The signal is set to "ON" even when there is no communication (logical 1). It should be remembered that this line signifies the input for the DCE (the peripheral), therefore it will be linked to the RXD signal. The signal used to receive the data is called receive data (RXD) (bits). On the opposite side, it is linked to the TXD. RTS is a control line used to request the bus for the transfer of data and to interact with the opposite side.

An extra control line called "clear to send" (CTS) is used to tell the other end that the device is prepared to receive the message. The hardware flow control makes use of the two lines (RTS and CTS). One device must increase the RTS signal in order to send data. When it is prepared to accept the data, the device on the other end will raise the CTS signal. The transmitting device won't begin delivering data until it receives the CTS from the other side. The opposite end device is still busy and unable to take the data if the CTS is not set. When the system provides a huge file to be printed, for instance, these two signals are used in a simple example. In most cases, the printer receives the data, buffers it, and then sends a print command. When a problem arises (paper jam, running out of paper, etc.), the printer must notify the other end that it can no longer receive new data. This is accomplished by not

raising the CTS or responding to the RTS. The opposite side is aware that there is a difficulty preventing the transmission and is waiting for the clear signal to come through. A bit called "data set ready" (DSR) indicates that the device on the other end is prepared (powered on, connected to the bus, and ready). As the device on the other end is not connected, the signal will be silenced if the phone line cuts off.

A signal line called Ground (G) is utilised to synchronise the devices on both ends. This signal is often wired to the PGND. A control line called carrier detect (CD) alerts the device that it has located its opposite on the other end. This bit is set when the device on one end is able to connect to the fax machine on the other end, to use a fax machine as an example. This adds a further level of communication. The telephone line confirms that the contact was established, but in the case of the fax, it's possible that a person, not a fax machine, is on the other end. Despite the fact that the link was established, it is insufficient for data transmission. Similar to this, when a transmission occurs between computer systems, the CD is activated when a connection is made and the recipient is able to comprehend and reply.

A control line called "Data Terminal Ready" (DTR) informs the peripheral device that it is ready to start a conversation. This signal is comparable to the DSR, however the DTR is linked to the distant side of the communication device, whilst the DSR is connected to the local side. A control line known as the "ring indicator" (RI) alerts users when their communication equipment detects an incoming call (the other side is initiating a communication). This signal, like a regular telephone, is what makes the phone start to ring. The phone will not signify that the call has been established or that the person on the other end may begin speaking until we accept the call. The serial bus transmits one bit at a time, in contrast to the parallel buses explained at the beginning of the chapter, which transfer many bits throughout each cycle. Depending on the specification, the data is sent over the TXS channel in groups of bits (either seven or eight bits at a time). If you want to transmit seven bits, for instance, you would send the first bit on the first cycle, the second bit on the second cycle, and so on until the whole block has been delivered. According to custom, the signal is set (logical "one" when there is no transmission).

On the serial bus, each group is transmitted with a start bit and a stop bit at the beginning and end, respectively. The start bit, which is a logical "zero," indicates that the data is in the bits after it. The block ends when the stop bit, which is a logical "one," is set. As will be discussed later in the chapter, extra parity bits are often supplied to the transmission to safeguard its integrity. Serial communication is seen in Figure 15.4 with the character "a" being sent. The least important bit is sent first, and its binary value is 1100001. The start bit, which signifies that the following bits in the transmission constitute the data, is sent first in a transmission. The data bits are delivered in the cycles that follow the start bit. The first bit to be delivered is a "one," which will be followed by four "zeroes," two more "ones," and then the transmission begins with the least important bits. It should be remembered that there are many communication settings available. The size of the byte is one of them. Either a 7-bit or an 8-bit byte may be used. Alternately, the parity might be defined as even or odd parity (to be detailed later in this chapter). The letter "a" is represented by seven bits in this case since the transmission parameters were set to seven bits bytes. The next bit communicated is a parity bit (which will be described later), and the last bit (the stop bit) indicates that communication has ended.

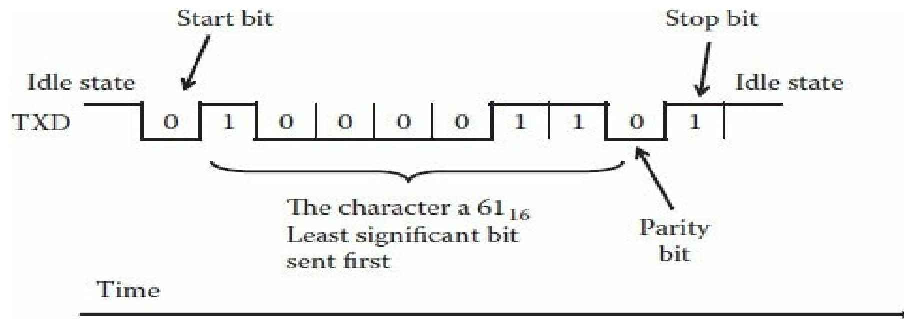

Figure 15.4 Serial communication.

The RS-232C serial bus was supplanted by USB, which has since become the de facto industry standard, even if it still has certain applications. Its popularity is due to the fact that it supports a wide range of devices in addition to computer peripherals, including cameras, mobile phones, media players, and more. Moreover, some of the gadgets with reduced power consumption are powered via USB as well. Owing to its popularity, a number of variations were created, most of which focused on performance improvements.

Development of the Bus Concept

The majority of contemporary commercial systems are built on PC architectures, which implies that the fundamental parts are those found in PCs. A hierarchy of buses is utilised because, even with PCs, the buses can cause bottlenecks. Even the bus hierarchy may not always be enough in big parallel systems with several PCs sharing a shared memory, as can be demonstrated.

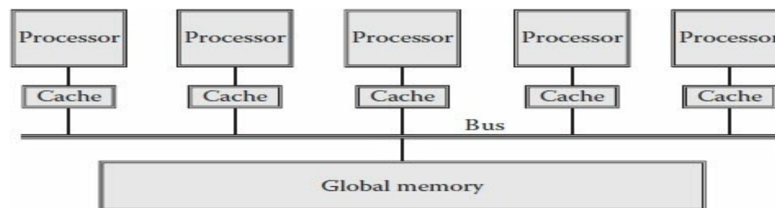

Figure 15.5: An example of a parallel system.

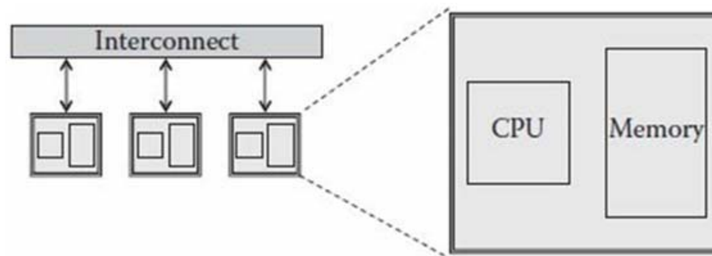

Figure 15.6: Cray T3E architecture.

The structure of the Cray-designed and -produced J90* supercomputer. The system has a number of processors (8, 16, or 32), each with its own cache memory. A private internal fast bus connects each cache memory to the relevant CPU. The system features a shared global memory that all of the processors may use to share applications and data. A quick bus (1.6 GB/sec for each CPU) was used for communication between the global memory and the processors. Despite the high transfer rate, it was clear that the bus would become a bottleneck

for bigger systems with more CPUs. Because of this, Cray's next supercomputer, the T3E, used a different strategy. The T3E came in a number of variations and utilised the Alpha chip, which was a highly fast floating-point calculator. The entry-level model of the T3E featured 1480 processors, and it was designed to support numerous processors. Moreover, it was the first machine to achieve one Teraflop (1012 floating-point calculation per second). Its architecture has to be changed in order to attain these speeds. The redesigned architecture is partially shown in Figure 15.5 and 15.6.

The system was built on isolated computing units in order to accommodate the high level of parallelism and get beyond the inherent constraints of buses, even the highly fast ones (processing elements [PE]). Each of these components is a standalone system with a CPU and memory (it has its own cache memory as well, but this was not included in the figure for simplification reasons). A new communication grid was created to link the PEs and the global memory. The grid is based on several channels, where each PE is linked to a few other PEs and the memory via various channels, in contrast to the bus, which is based on one channel that links all the devices. The architecture used was based on the geometric form of a torus in order to make implementation simpler. The topology's key feature is that there are no PEs at the ends and that every PE is linked to precisely four neighbours. A bus-based design for linking several computer units is shown in Figure 15.7. Eight distinct buses are shared by 16 units in the diagram. The bus design depicted on the right side of Figure 15.8, which is built on a grid with each PE connecting to every other PE, is quicker. However when there are a lot of components that need to be linked by this kind of grid, it becomes quite difficult to construct the grid since there are so many connections that need to be made. Remember that the formula for the number of connections needed for n items is:

$$\textbf{Number of connections} = \textbf{n*}\ \frac{\textbf{(n-1)}}{\textbf{2}}$$

So, for a 1480 components and a global memory, as designed for the entry level T3E, the number of channels required will be 1,094,460

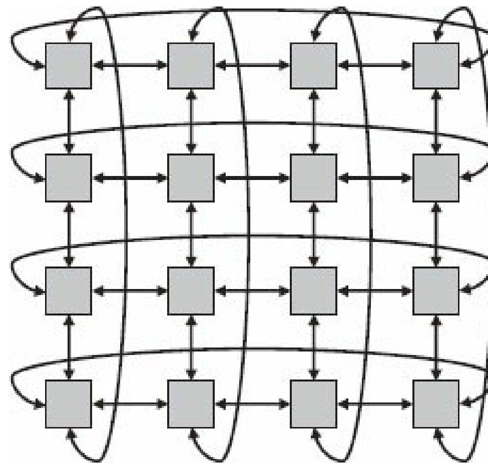

Figure 15.7: a torus-based bus.

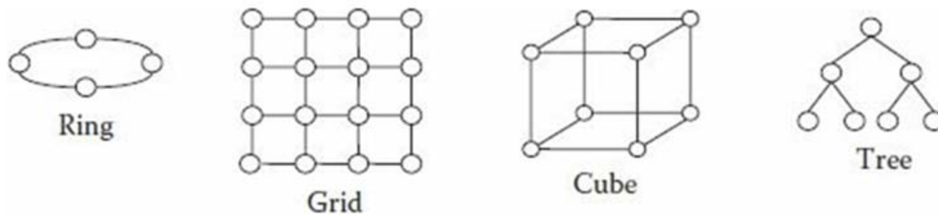

Figure 15.8 Additional topologies.

Of fact, there are many alternative topologies that may be used to create a parallel system. The choice should ultimately balance the expenses with the requirements (necessary transfer rate, bus length, dependability, survivability, etc.). Even in the case of a bus failure, the ring, grid, and cube topologies all have some level of resilience. With any circular topology, access to all the components is still feasible even if a portion of the network is offline. In this situation, it is claimed that the network (or the topology used to build the network) has built-in survivability. In Figure 15.9 right-hand tree architecture, one or more components will become unreachable if only one route fails, hence there is no provision for survivorship. Due to the near closeness of all components, there is extremely little probability that a bus-related issue would arise in a parallel system. Yet, distributed systems, even those that are far placed, may use the same kind of connection. Given that Internet connection will often be used, the Internet's rapid expansion over the last ten years has considerably diminished the significance of distributed systems topology. Network topologies are still useful and significant for more crucial systems where uptime is a key aspect.

The cube offers a straightforward technique for choosing the best path, as do any other topologies where there are several ways to travel from one component to another. By assigning binary integers to each node, this is accomplished. In this method, the node's number differs from the numbers of its neighbours by only one bit.

Outside the System Limits Bus Expansion

The bus was described at the beginning of the chapter as a way to link the system's different parts together. In actuality, the bus was mostly utilised by computers in the 1970s and 1980s to link the system's internal parts. As a result, the bus's length was very constrained, only reaching a few feet at its greatest. The buses required to be longer as the systems developed, for connecting a printer that was in a separate room, for instance. Further advancements, like as the use of widespread storage arrays, necessitated lengthening the bus by up to 30 feet (Figure 15.10). The 1990s are known as the "Networking Era." Even Sun Microsystems said that the network is the computer (see the chapter 1 part titled "The Network is the Computer" for more information). Modern systems allow users to operate on virtual systems that may have a number of servers, each of which is devoted to a separate computing task. The person using the system has no idea which system they are using or where it is situated, and they certainly don't care. The user's only concern is receiving the requested service. A collaborative virtual system is created by the organisational network and the systems that are linked to it in this way. The 1990s saw the beginning of this form of operation, which coincided with the growth of LANs. The LANs' initial length was less than a mile, but with the help of numerous repeaters and boosters, it was increased to around a mile and a half.

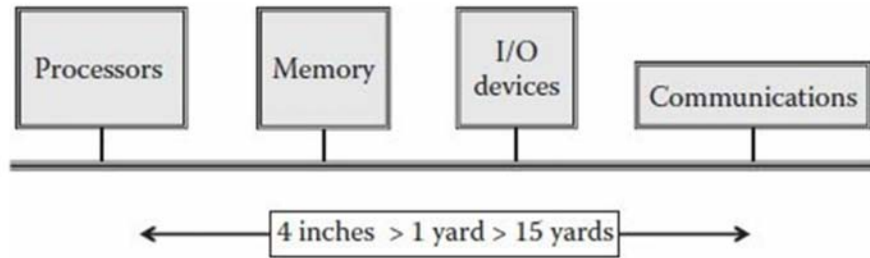

Figure 15.9: Bus length over the years (1970–1990).

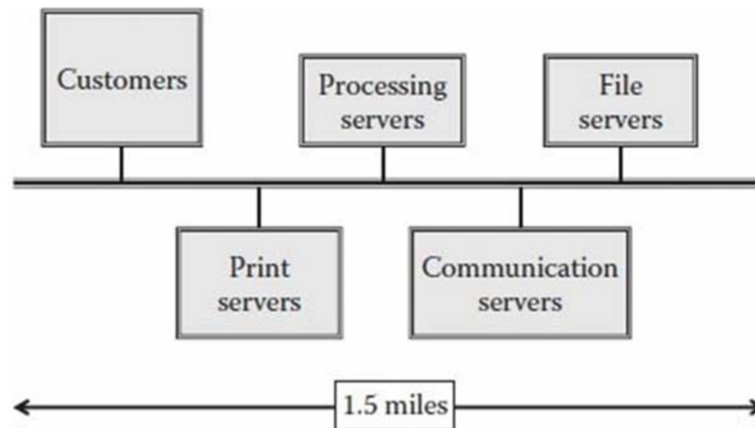

Figure 15.10: 1990 bus (local area network).

Early in the year 2000, the Internet quickly replaced many of the antiquated computer ideas. The computer business has transformed as a result of faster data transfer and a wider range of communication options. As distance is no longer an issue, many organisational systems are now dispersed around the world. Additionally, by contracting out a bigger portion of the applications and the infrastructure that supports them to other committed and knowledgeable firms, cloud computing technologies are gradually changing the local computer division. In terms of bus length, it indicates that the new "buses" are now endless (Figure 15.11).

Elements of Reliability

Due to the rapid growth of different buses, some of which are extensively used, dependability issues needed particular attention. Since the distance and bus length were less in the past, it may have been reasonable to expect that if the wires weren't severed, the bus would keep running and the data would be safely conveyed. The situation is different right now. While the computer's buses are very dependable, dependability must include all components and not just the system as a whole. Data is often sent through wide-area networks that use various transmission mediums; some of them are based on wireless communications, which may break down owing to the location of the device.

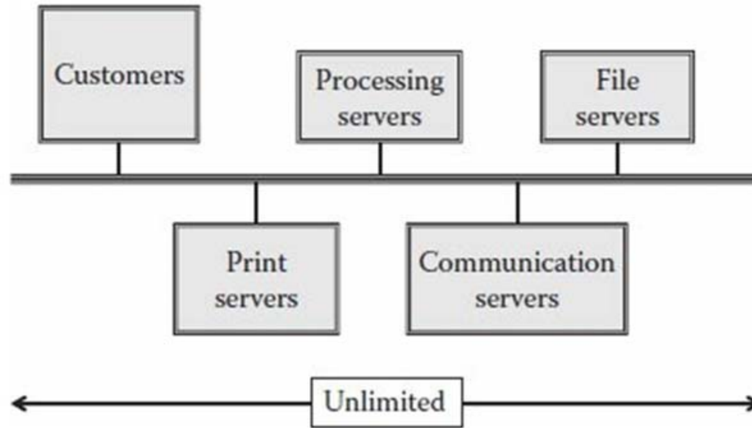

Figure 15.11: The 2000 bus (Internet).

Throughout time, numerous techniques for improving transmission dependability were developed. While the techniques are discussed as part of this chapter on buses, it should be emphasised that they may also be used with other devices, such as disc drives. While using a disc, the controller has to confirm that the data it receives is exactly what was written and that no bits were altered in the process. Many techniques for safety and greater dependability have been created throughout time. The simplest approach relies on error correcting coding (ECC). There are several ways to implement ECC, but in each instance, the goal is to add a few extra bits to the data being communicated so that the receiving end may verify that the transmission was correctly received. An established method is used to compute the bits that must be added. The procedure is then used to compute the ECC at the receiving end. If a match is made, no error was identified, indicating that the data block transmitted and received are the same. The block may be repaired (using specific techniques) or resent if the computation, on the other hand, indicates some discrepancy.

A parity bit is the foundation of the most fundamental and simple approach. The parity may be even or odd. Prior to transmission, both parties must be aware of the parity type (odd or even). The process is based on adding the required parity bit to the transmission block after counting how many bits in the transmission are set (one). The amount of bits set in the transmission block, including the parity bit itself, will be an even number if the technique uses even parity. The parity bit is always in charge of ensuring that the overall amount of bits being set, including the parity itself, complies with the specification (it is either an odd number or an even number).

Consider the following 8-bit byte, which is to be conveyed using even parity in the transmission:

D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	1	1	1	0	1	1	0

As the communication is based on even parity and the number of bits set is odd, the parity bit should be set. As a result, there will be an even number of bits transferred overall, including the parity bit itself. The transmitted bit block will be

D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	P
0	1	1	1	0	1	1	0	1

The parity bit is the bit on the right-hand side. The block that was received must be verified at the receiving end. The agreed-upon algorithm is used to do this. The amount of bits that are set in this particular instance will be tallied, and if the result is an even number, it is assumed that the transmission was received without any problems.

Alternatively, if the block that was received included

D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	P
0	1	1	1	0	1	0	0	1

The receiving end recognises that this is an odd number after counting the bits that have been set. This indicates that something went wrong during the transmission since the block received is not the block transmitted. This straightforward approach just indicates the mistake; it does not provide a way to fix it. The receiving end will have to ask the sender to transmit the block again in this particular situation. The parity approach can only identify problems if one bit was modified, or alternatively if an odd number of bits were changed. The procedure will continue to believe the block received is valid even if two bits or any other even number of bits were modified.

We'll take the block from the previous example as an example once again. In this instance, the approach employs odd parity to ensure that the parity bit is clear:

D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	P
0	1	1	1	0	1	1	0	0

Unfortunately, the block received was,

D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	P
1	0	1	0	1	0	1	1	0

The receiving end counts the bits transferred and notices that there are an odd number of bits, therefore it deduces that the block was successfully received. Sadly, there are relatively few similarities between the two blocks. While this is a made-up example, it was selected to highlight the method's flaws. The fact that the transmission block often comprises many bytes is the basis for a relatively straightforward and efficient solution to the issue posed by the previous example. This implies that both the horizontal and vertical applications of parity are possible. The vertical check will be accomplished by including an extra parity byte, whereas the horizontal check was given by the prior technique. In this example, we'll suppose that an odd parity type will be applied. Let's say we need to convey a block of 8 bytes that is specified as follows:

D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	1	1	1	1	0	0
0	1	0	1	1	1	0	1
1	1	0	0	1	1	0	0
1	1	0	1	0	1	1	1
1	1	0	1	1	1	1	1
1	1	1	1	0	1	1	1
1	1	0	1	1	1	0	0
1	0	0	0	0	0	0	0

The parity bit for each byte will be determined in the first phase. The vertical parity byte is computed after all parity bits have been determined and added to the data bytes. The leftmost parity bit is computed based on the leftmost bits of all the bytes in the block and will once again be of the odd parity type. The parity byte's other bits will also be determined in a similar manner. The block won't be sent until after the new parity byte has been inserted.

D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	P
0	0	1	1	1	1	0	0	1
0	1	0	1	1	1	0	1	0
1	1	0	0	1	1	0	0	1
1	1	0	1	0	1	1	1	1
1	1	0	1	1	1	1	1	0
1	1	1	1	0	1	1	1	0
1	1	0	1	1	1	0	0	0
1	0	0	0	0	0	0	0	0
1	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**0**

The new parity byte makes sure that there are always odd numbers of bits set in each column.

Without having to retransmit the block, this system of vertical and horizontal checks may identify a bit that has changed and even fix it. The technique may be able to fix things even if more than one bit was altered during transmission (based on the location of the changed bits). On the other hand, the issue with this approach is that the check must wait until the whole block has been processed. Large buffers for storing the whole block are implied when the tests are performed by the hardware. The extra buffers don't contribute much in terms of pricing right now since memory costs are so cheap. But, the scenario back then was different, which prompted the sector to look for more affordable options. The overhead related to the approach is another aspect that should be taken into account. Additionally, the transmission overhead increases with block size. The overhead is fixed by the parity bit. An extra bit is added to each byte of 7 or 8 bits. This amounts to a 1.3% cost (for 8 bits) or a 12.5% overhead (for 7 bits). The overhead associated with adding the parity byte might be quite large. The parity byte adds 100% overhead on top of the previously mentioned parity-bit cost if the blocks being transferred are one bytes in size. The blocks that are sent are often big to reduce the overhead. The parity byte, for instance, will only add a 5% cost to a 20-bit block.

Nevertheless, under these circumstances, the whole block must be retransmitted if an error occurs and the approach is unsuccessful in fixing the issue. The operating system may employ a variable length block in more advanced ways on modern computers. In these circumstances, a set of factors determines how the block length is modified. This, however, has to do with specific applications that the operating system supports and is not covered by this explanation.

It should be emphasised that many times, as has previously been shown in this book, the solutions utilised today were created to address an issue that, as a result of fast technical breakthroughs, no longer exists. Yet, we appreciate these answers as they stand or as a foundation for a superior solution that was added to the original. Based on checksum, there is another technique for verifying the block that was received. There is no extra bit in this instance (like the parity bit). Instead, one extra byte is included. Assume for the moment that we must send a 12-byte chunk. Let's also suppose that three-byte chunks are used for the checksum calculation. The first three bytes will be added by the algorithm, and any overflows will be disregarded. The algorithm will then send the first three bytes, followed by a byte that provides the determined checksum. Although there are 12 bytes to be communicated in this case, the procedure will repeat four times. The identical checksum is computed on the receiving end, and if the results match, the block was successfully received.

Questionnaires

1. What do you understand by the term Computer Architecture?
2. Is Computer Architecture different from a Computer Organization?
3. Do you know the basic components used by a Microprocessor? Explain.
4. What are the various Interrupts in a Microprocessor system?
5. What are the common Components of a Microprocessor?
6. What do you know about MESI?
7. Are you aware of Pipelining?
8. What do you know about Cache Coherence?
9. What do you know about Virtual Memory?
10. What are the 5 stages of the DLX pipeline?

References Books for Further Reading

1. "Computer Fundamentals Architecture and Organization" by Ram B
2. "Fundamentals of Computer Organization and Architecture (Wiley Series on Parallel and Distributed Computing)" by Mostafa Abd–El–Barr and Hesham El–Rewini
3. "Fundamental of Computer Organization and Design" by Sivarama P Dandamudi
4. "Fundamentals of Computer Organization and Architecture" by Jyotsna Sengupta
5. "Creating a Data–Driven Organization" by Carl Anderson
6. "Structured Peer-to-Peer Systems: Fundamentals of Hierarchical Organization, Routing, Scaling, and Security" by Dmitry Korzun and Andrei Gurtov
7. "Computer Fundamentals, Third Edition: Architecture and Organization" by Ram B