International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 08 Issue: 03 | March - 2024 SJIF Rating: 8.176 ISSN: 2582-3930

Instruction Level Parallelism and Memory Synchronization

Rupam Sardar

Budge Budge Institute of Technology

Abstract:

The simultaneous or parallel execution of a series of instructions within a computer program is known as instruction-level parallelism, or ILP. ILP stands for the average number of instructions executed throughout each stage of this parallel execution, to be more precise.

The architecture known as instruction level parallelism (ILP) allows for the execution of several operations in parallel within a single process, each with its own set of resources, including address space, registers, identifiers, state, and program counters. It describes compiler design strategies and processors intended to carry out operations in parallel to increase processor performance, such as memory load and store, integer addition, and float multiplication.

SUPER SCALAR PROCESSOR

A more aggressive approach is to equip the processor with multiple processing units to handle several instructions in parallel in each processing stage. With this arrangement, several instructions start execution in the same clock cycle and the process is said to use multiple issue. Such processors are capable of achieving an instruction execution throughput of more than one instruction per cycle. They are known as 'Superscalar Processors'.

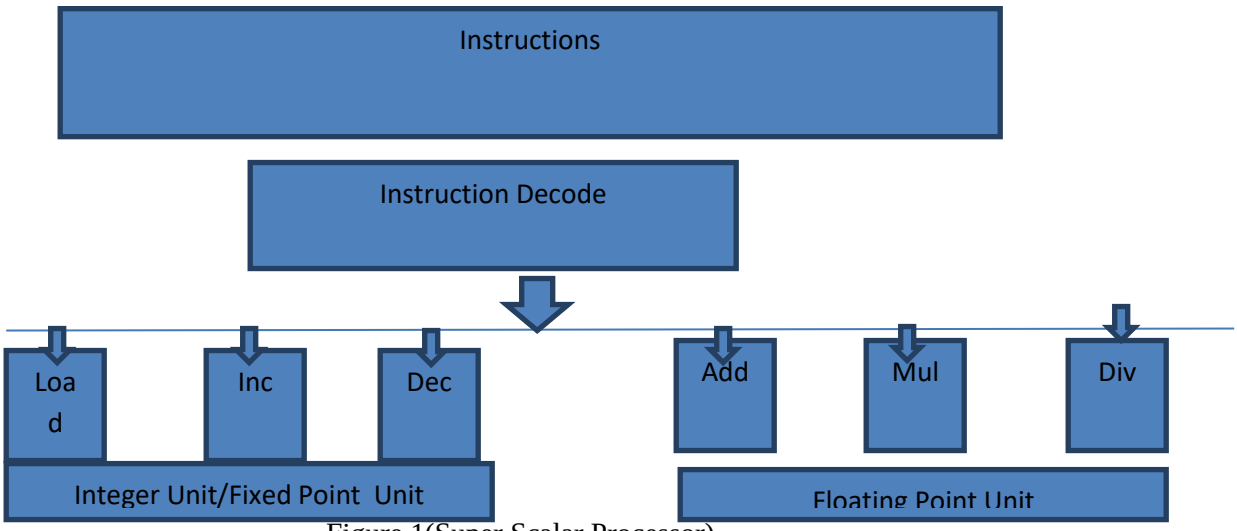


Figure 1(Super Scalar Processor)

Beyond what simple pipelining can provide, three techniques have been developed to boost performance: Superscalar, Super pipelining, and Very Long Instruction Word (VLIW). By breaking down a pipeline's lengthy latency phases—like the memory access stages—into multiple shorter stages, super pipelining increases performance and may increase the number of instructions that run in parallel during each cycle.

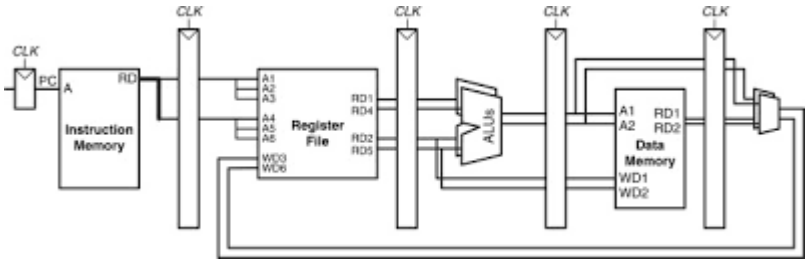


Figure 2: Super Scalar Processor

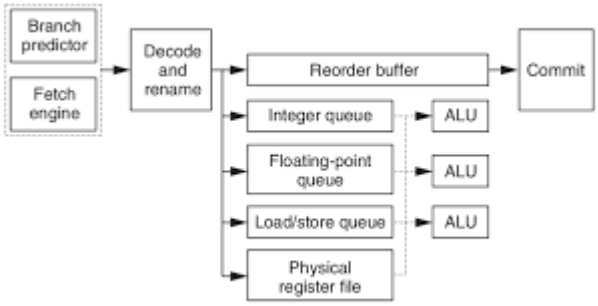


Figure 3:Internal Diagram of Superscalar Processor

Very Long Instruction Word:

VLIW techniques, on the other hand, enhance performance by sending out several instructions in a cycle. Every cycle, VLIW statically issues several instructions, while Superscalar dynamically issues numerous instructions. These methods were first (and currently serve as the foundation for) contemporary computer architecture designs. In this chapter, different approaches are described, instances are examined, and their impact on the designs of contemporary microprocessors is talked about.

Techniques like Superscalar, Super pipelining, and VLIW were created to enhance performance above and beyond what simple pipelining could provide.

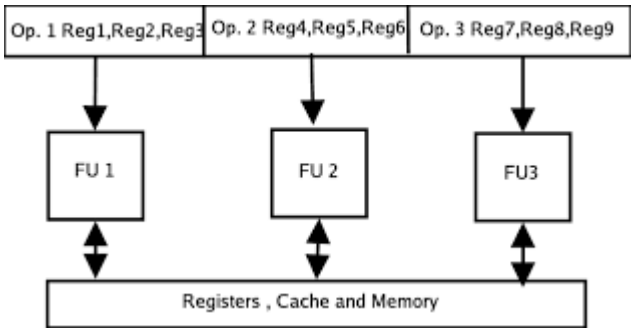


Figure 4: VLIW Architecture(Basic)

By breaking down a pipeline's lengthy latency phases—like the memory access stages—into multiple shorter stages, superpipelining increases performance and may increase the number of instructions that execute concurrently during each cycle.

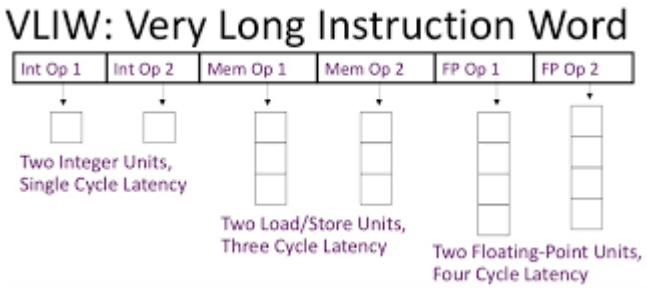


Figure 5: How VLIW Execute

Superscalar and VLIW techniques, on the other hand, enhance performance by sending out several instructions in a cycle.

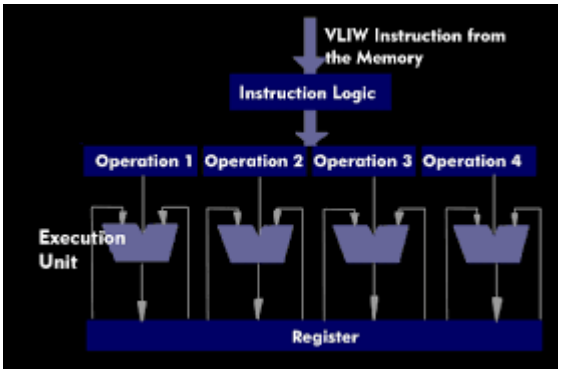


Figure 6: VLIW Execution Unit

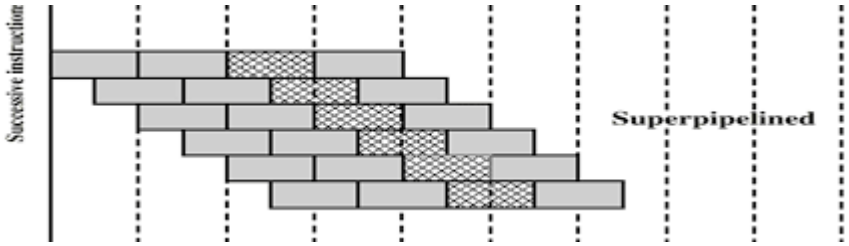
The number of instructions in the program at hand is denoted by IC in this equation, and the average number of clock cycles per instruction is denoted by CPI. The length of a clock cycle is represented by clock_cycle_time.

Superpiping.

Of the three, superpipelining is the most straightforward. Unlike superscalar architectures, it doesn't require extra hardware (such functional units and fetch units). It additionally does Superscalar Structures Utilizing a high degree of Instruction Level Parallelism (ILP) is one of the primary design challenges for today's high-performance microprocessors. The ILP is the main technique for increasing a sequential code's parallelism. Out-of-order execution and the multiple instruction problem are the two main techniques for taking advantage of instruction level parallelism. In order to carry out as many instructions as feasible inside a single clock cycle, those two strategies were developed. Furthermore, advanced branch prediction

Superpiping

All superpipelining does is raise the clock rate at which the pipeline runs on a CPU to increase performance. The faster clockVLIW is designed to take advantage of instruction-level parallelism by employing a lengthy instruction word with several fixed-number operations. These operations don't pose a risk to data or control when they are fetched, decoded, issued, and carried out simultaneously. Consequently, every operation contained in a single VLIW instruction needs to be completely independent.

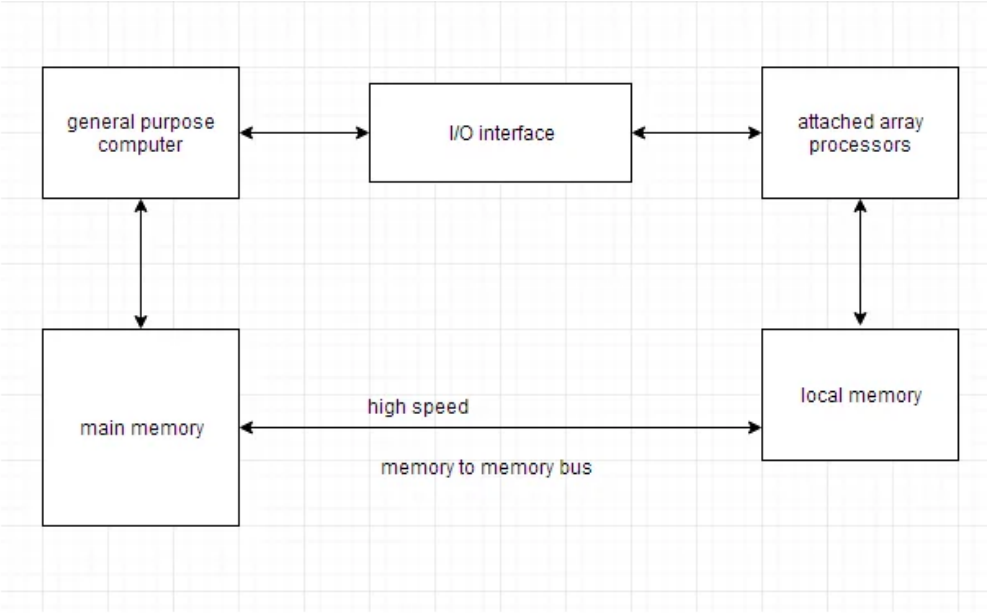


Types of Vector(Array) Processors

Multiprocessors and vector processors are other names for array processors. They work with vast amounts of data to do calculations. They are therefore employed to raise the computer's performance.

A general purpose computer can perform better while performing numerical computing tasks when it has an attached array processor, which is a processor that is attached to the computer. It processes many functional units in parallel to obtain high performance computer.

Array Processors Attached
Simple Array CPUs
Array Processor Affixed



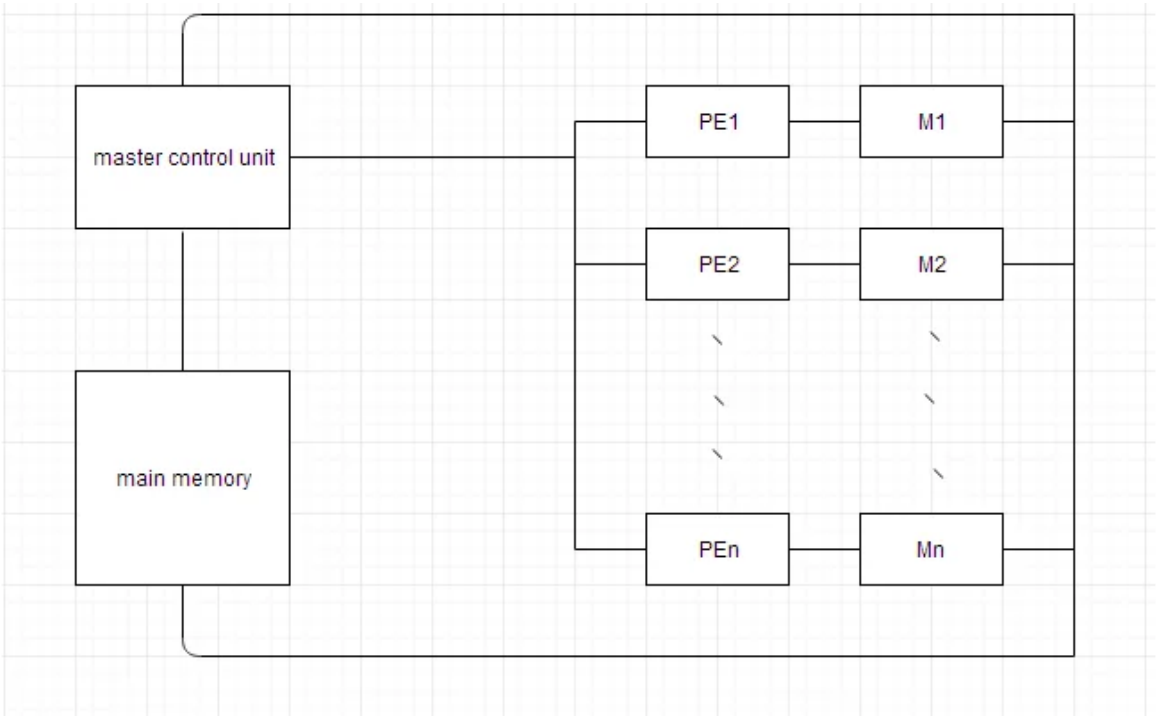
Simple Array CPUs

A single computer with many processors running in parallel is referred to as a SIMD. A single instruction stream and several data streams are provided by the processing units' design, which places them under the direction of a single control unit.

Below is a general block diagram of an array processor. It has a group of processing elements (PEs) that are all the same and have a local memory of M each. ALUs and registers are present in every processor component. Every operation of the processing elements is managed by the master control unit. In addition, it decodes the instructions and decides how to carry them out.

The software is kept in the main memory. Getting the instructions is the responsibility of the control unit. All PEs get vector instructions at the same time, and the results are sent back to memory.

The Burroughs Corp.'s ILLIAC IV computer is the most well-known example of a SIMD array processor. Computers using SIMD processors are extremely specialized. They are not appropriate for other kinds of computations and are limited to solving numerical issues that can be represented as vector or matrix forms.

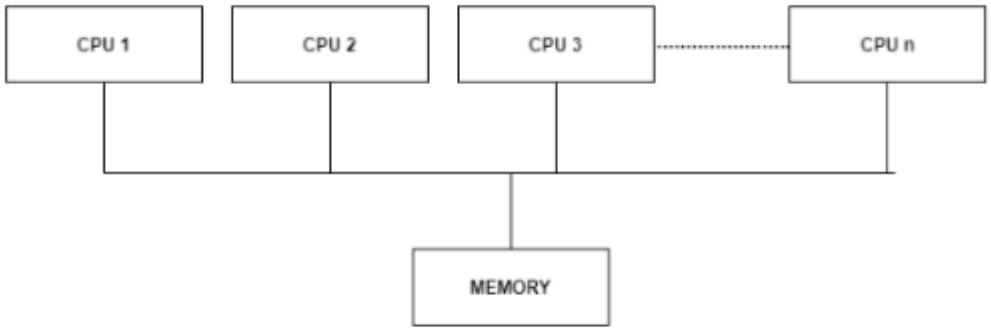


The Array Processor: Why Use It?

The utilization of array processors enhances the speed at which instructions are processed overall. Additionally, since the majority of array processors function independently of the host CPU, the system's overall capacity is improved. • Because array processors have their own local memory, they can give systems with minimal memory additional memory.

A system with numerous processors and a means of communication between them is called a multiprocessor system. Homogeneous multiprocessing, commonly known as symmetric multiprocessing (SMP), is a popular type of multiprocessing in computer systems where two or more identical processors share a single main memory.

The majority of computer systems only have one processor, or are single processor systems. Parallel or multiprocessor systems, however, are becoming more and more significant these days. These systems feature numerous processors operating in parallel that share peripheral devices, memory, the bus, and the computer clock. An illustration of the architecture with several processors is



Multiprocessing Architecture

A centralized architecture is one in which all nodes are linked to a central coordination system, which will share any information that the nodes want to share.

Distributed Systems with a Centralized Architecture

A centralized architecture is one in which all nodes are linked to a central coordination system, which will share any information that the nodes want to share. A centralized design prefers that most components are clustered together and not duplicated elsewhere, unlike a distributed architecture, which would need all functions to be in a single location or circuit.

It includes the subsequent categories of architecture:

Layering Client-Server Applications

Customer-Server
The core client-server architecture of a distributed system divides processes into two (possibly overlapping) groups. A program that offers a specific service, like a file system or database service, is called a server. A client is a process that makes a request to a server and then holds off on making a request for a service until the server responds. The image below illustrates this client-server communication, commonly referred to as request-reply behavior:

A simple connection-less protocol can be used to establish communication between a client and a server when the underlying network is fairly reliable, as it is in many local-area networks. In these cases, when a client makes a service request, they merely package a message for the server that includes the necessary input data and the service they desire. The message is then sent to the server. Conversely, the latter will consistently await incoming requests, handle them, and then compile the results into a reply message that is sent to the client.

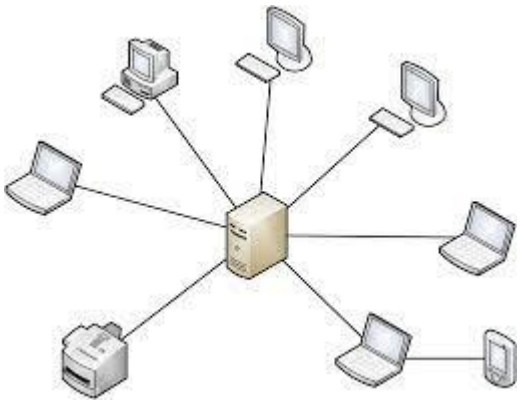


Figure 8: Customer Swerver

One obvious advantage of utilizing a connectionless protocol is efficiency. The request/reply protocol that was just outlined functions as long as there are no lost or damaged messages. Unfortunately, strengthening the protocol to withstand sporadic transmission faults is not a simple task. The only thing we can do if we don't get a reply message is let the client try again with the request. The client's capacity to ascertain whether the reply's transmission succeeded or failed, or whether the initial request message was misplaced, is problematic.

Many client-server systems have an alternative in the form of a trustworthy connection-oriented protocol. This approach works best in wide-area networks when communication is intrinsically unstable, but it is not entirely appropriate for local-area networks due to its relatively low performance.

Layering Applications

However, as many client-server applications are designed to offer users access to databases, many have pushed for a differentiation between the three levels below, thereby following the layered architectural approach we previously outlined:

The degree of the user interface
The degree of processing
The degree of data

The user interface level has all the necessary components to establish a connection with the user, including direct display management. At the processor level, applications are frequently seen. At the data level, the real data that is utilized to inform decisions is controlled.

The Level of the User Interface

Clients are typically the ones who implement the user interface level. This level consists of programs that allow users to interact with apps. The degree of complexity varies greatly throughout application packages. The simplest user interface program is one that uses a character-based screen.

This type of interface has generally been used in mainframe environments. In scenarios where the mainframe

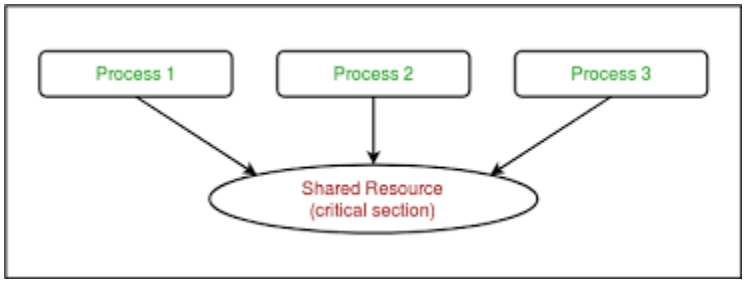
controls all aspects of interaction, including the keyboard and monitor, one hardly ever talks about a client-server architecture.

The Degree of Processing

The middle section of the architecture is this. This makes sense as it is the area of the system where all processing is done at the level of user interaction and data. It carries out certain tasks and processes the requests made through the user interface.

The Level of Data

The programs that manage the real data that the applications run on are located at the data level in the client-server architecture. Data at this level are frequently persistent, meaning they are kept somewhere for future use even when no application is open. This is an important characteristic.



Think of a search engine on the Internet. A search engine's user interface is quite straightforward: after entering a string of keywords, the user is shown a list of Web page titles. An enormous database of prefetched and indexed Web pages makes up the back end. A program that converts a user's string of keywords into one or more database queries is the search engine's central component. The results are then ranked and turned into a list, which is then converted into a collection of HTML pages. This information retrieval component is usually positioned at the processing level in the client-server architecture.

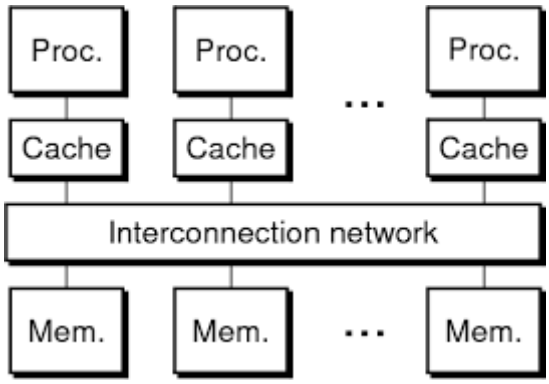
In summary

In conclusion, centralized architecture simplifies management and improves security, but it might not be the ideal option for applications requiring fault tolerance, high performance, or scalability.

Shared Memory Synchronization

Abstraction. In computer science, synchronization is a fundamental problem. It is quickly becoming as a significant performance and design concern for distributed and concurrent system design, as well as concurrent programming on contemporary architectures. In order to give the reader a sense of the nature of this fascinating and significant topic, I have attempted to progressively introduce the reader to some of the most fundamental problems and classical results underlying the design of concur-rent systems in this overview. The subjects discussed are a subjective sampling of

the highlights in the field of concurrent programming and synchronization techniques rather than a complete survey of it. A detailed discussion of every outcome addressed here, along with many others, can be found in [34].



One of the key innovations that is fundamentally and irreversibly altering the computing landscape is the mainstream computer design that incorporates several processors into a single device. The "brain" of a computer is its processor, and modern, popular computers are constructed with several "brains."

The way applications are designed for computers with many processors must fundamentally change in response to these fundamental changes in computing architectures. Fundamental problems like concurrency and synchronization are vital to the design of these kinds of systems. How skillfully programmers use concurrency will determine a lot about the future of multiprocessor machines.

Two primary technological pillars support the amazing, swift advancements in multiprocessor computer technology. Naturally, the first is the development of speedier hardware design. The creation of effective concurrent algorithms that enable intricate processor interactions is the second.

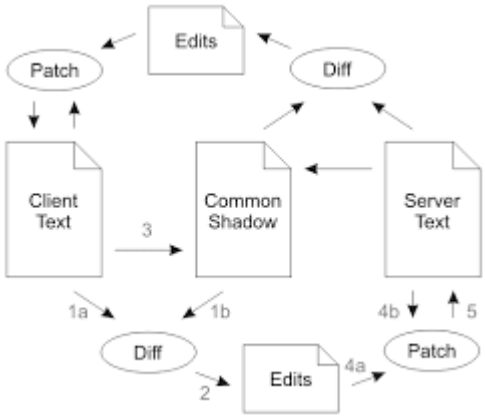
A concurrent algorithm is the formula that many computer elements use to solve a problem. Concurrent programming was first published in a work by Edsger Wybe Dijkstra in 1968 [10]. I've made an effort to progressively introduce the reader to some of the ideas and findings that underpin the creation of these concurrent algorithms in this survey. Details on every outcome described here, along with many more, may be found in [34].

Mainstream computers were constructed with a single CPU only until a few years ago. As it becomes harder to raise the (clock) speed of uniprocessor computers, the situation is rapidly changing. As a result, every microprocessor manufacturer is now compelled to stake their futures on having several processors—also known as multicores—installed in a single computer.

These days, the majority of computer manufacturers provide a new generation of multiprocessor computers, which include multiple complex processors that work together to do multiple tasks concurrently on a single computer. For many years, a number of computer manufacturers have been producing extremely costly high-end computers with multiple processors. However, it is only recently that comparatively inexpensive multiprocessor PCs have become widely accessible and are currently present in many homes.

Synchronization

Syncing is a common element of our everyday interactions with other individuals. It could be necessary for you and your spouse to coordinate who gets to buy groceries, takes out the trash, drops the kids off at school, takes the car, uses the one computer you have, and who showers first (if you have a single shower at home). If you and your neighbor share a yard and your neighbor has a dog and you have a cat, you might want to work together to make sure that neither pet is ever in the yard at the same time.

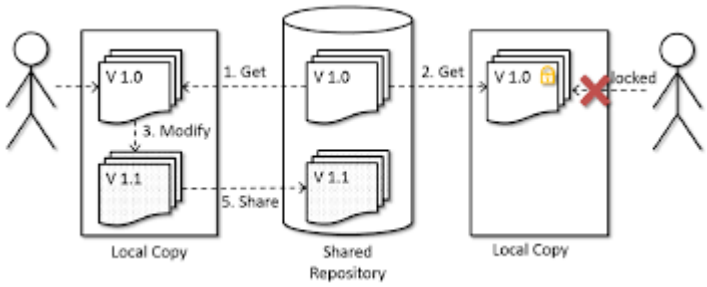


In each of these instances, synchronization is employed to guarantee that only one player—and not both—will perform a specific action at a specific moment. Cooperation is the subject of another kind of synchronization. A hefty table may need to be moved by you and your spouse because it is too big for one person to handle alone. Determining the precise moment for a coordinated attack on the enemy camp by two camps within the same army is a classic example of collaboration.**Models of**

Concurrency

A group of processors that converse by reading and writing to a common memory is known as a concurrent system. A process and a certain computation match. In other words, the execution of a program is a process. A processor is the actual hardware that powers a process. On a single processor, multiple processes can operate simultaneously, but only one of them can be active at any given time. When multiple processes are operating simultaneously on multiple CPUs, real concurrency is attained.

Multiple threads can be found within a single process. For instance, a process that synchronizes with the way a game is played could consist of three threads: the first handles keyboard commands.



the third person is in charge of I/O, the second person does computations. All of the findings apply to both processes and threads, however we will only be using the concept of a process.

There are two types of interactions that occur between processes or processors (we will refer to them interchangeably): synchronization and data communication. The interchange of data, whether by message transmission or reading and writing from shared memory, is known as data communication. When many processes must follow certain order constraints, synchronization is necessary and can be categorized as either cooperation or contention.
2. For instance, if Alice and Bob were to rearrange the order and first leave a note, then check the refrigerator door, the answer would be off.

Atomic Functions

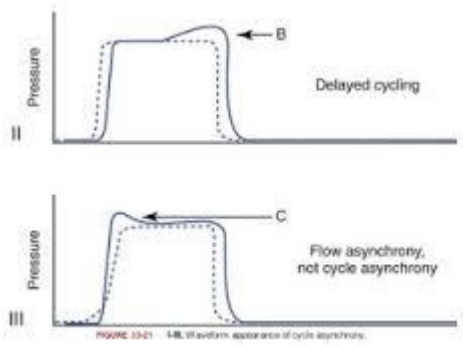
The design used by the majority of concurrent algorithms assumes that processes interact asynchronously through shared objects. Every item has a category that specifies the collection of oper-

ations that are supported by the item. Additionally, each object includes a sequential specification that describes how it will act in a certain order when these operations are applied.

Asynchrony

When processes are asynchronous, there are no presumptions made about their respective speeds. The degree of atomicity supported by different architectures varies. Operations whose execution is unaffected by other concurrent actions are referred to as atomic (or in-divisible) operations. Atomic registers are shared objects that provide atomic read and write operations, and they are supported by all architectures.

– Swap: swaps the values of a local register (ℓ) and a shared register (r) atomically.
Gather-and-adjust: requires a register, r. After adding one to the value of r, the previous value of r is given back.
- Compare-and-swap: requires two values, new and old, along with a register, r. The value of register r is set to new and the value true is returned if the current value of the register is equal to old; if not, r is left unaltered and the value false is returned.



- Load-link/store-conditional: In this scenario, the store-conditional operation attempts to atomically put a value into register r, while the load-link operation atomically reads the value of the register r. Process p's store-conditional operation on r only writes a value if it hasn't been modified by any other process since p's previous load-link operation on r.

It yields a failure status otherwise.

- take-modify-write: A process can take a value from a shared register, compute a new value depending on the value, and then assign the new value back to the register in a single atomic action.

Although operations of concurrent processes may overlap, each operation should appear to take effect instantly. The concept of atomicity is overly stringent, and it is safe to loosen it by assuming that processes might attempt to access the object simultaneously. Specifically, non-overlapping operations ought to occur in their "real-time" sequence. This kind of linearizability criterion for accuracy is known [20].

Lock implementation is made easier by synchronization techniques like semaphores, which are typically implemented by modern operating systems. Modern programming languages, including Java and Modula, also include the monitor notion, which is a software module intended to guarantee resource exclusivity.
Remote vs. Local Memory Accesses

concurrently. Specifically, non-overlapping operations ought to occur in their "real-time" sequence. This kind of linearizability criterion for accuracy is known [20].

It makes sense to distinguish between local and distant shared memory access for some shared memory systems. Examine the next three architectures of machines:

1. Central shared memory systems, in which the shared memory is housed in a single central location and no process (or processor) has a private cache of its own. Because it must pass through the network of connections that connects the process to the memory, each access to a shared memory location in these kinds of systems is distant.

2. Cache coherent systems, in which every process has a personal cache of its own. A copy of a shared memory location is migrated to a local cache line when a process accesses it; this copy remains locally available until another process modifies the shared memory address, invalidating the local copy.

3. Distributed shared memory systems: in these systems, each process "owns" a portion of the shared memory and stores it in its own local memory as opposed to having the "shared memory" at a single central place. If a shared memory location is in the portion of the shared memory that is physically located on a process's local memory, then that process can access it locally.

It is imperative to make an effort to minimize the frequency with which a process must visit a shared memory location that is not within its local memory.

Mutual Exclusion Locks

The exchange of conflict between processes is what resource allocation is all about.

The challenge is in resolving conflicts that arise when multiple processes attempt to

to employ common resources. Stated differently, how to distribute common resources among competitors

Writing the entry and exit codes in a fashion that satisfies the next two fundamental requirements is the mutual exclusion problem. ing procedures. An instance of a broader resource allocation issue is the reciprocal

issue of exclusion in which there is just one resource accessible.

Building a mutual exclusion lock, or the mutual exclusion problem

guaranteeing exclusive access to a particular shared resource in the event of a problem

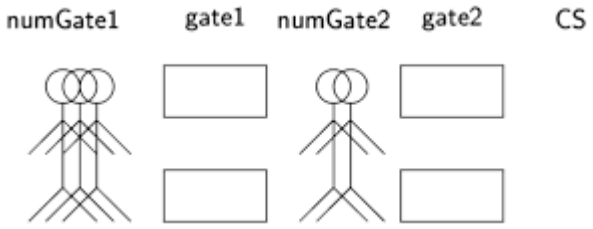
Multiple competing processes exist [9]. Operating systems, database systems, multiprocessor PCs, and computer networks are all affected by this issue. Given that it is the root cause of many interprocess synchronization issues, it is extremely significant.

Starvation-freedom:

If a process is attempting to reach its critical phase, it will inevitably reach its critical phase. Starvation freedom permits processes to execute their crucial portions arbitrarily many times before another trying process can run its critical section, despite the fact that it is strictly stronger than deadlock freedom. This is not allowed under the following condition for fairness.

FIFO, or first-in, first-out, A process that is already waiting for its turn to enter its critical section cannot begin its critical section earlier than any other process.

Deadlock freedom and mutual exclusion were the first two qualities that Dijkstra's original formulation of the problem required. These are the bare minimum standards that one could wish to set. It is assumed in the issue solving process that once a process begins executing its key part, it will always finish it, independent of what the other processes are doing. Among all the interprocess synchronization issues, the mutual exclusion problem has received the greatest research attention. This is a tricky problem that looks like it would be easy to tackle at first.

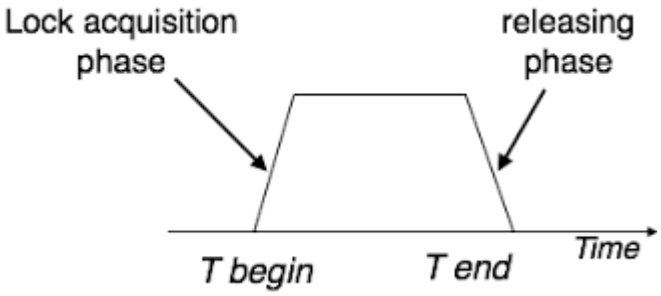


Parallel Data Structures

It is necessary to synchronize the concurrent access of several processes to a shared data structure to prevent interference from competing operations. Concurrent data structures can be built using a variety of techniques.

Lock-based Time Coordinating

The de facto method for concurrency control on concurrent data structures is a mutual exclusion lock, which allows a process to access the data structure only within of a critical section code that ensures it has exclusive access. The apparent simplicity of the programming model for these locks and the availability of scalable and effective implementations are major factors in the appeal of this method.



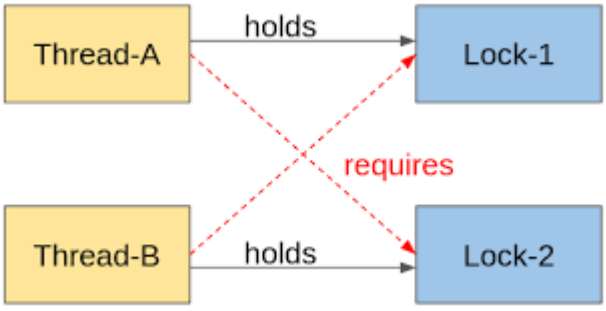
Because it makes processes wait for a lock to be released, using mutual exclusion locks to secure access to a shared data structure may negatively impact the performance of synchronized concurrent applications. Furthermore, processes that are sluggish or have halted could hinder other processes from ever accessing the data structure. Lock-free data structures can help prevent locking when simple data structures like queues, stacks, heaps, linked lists, and counters are updated concurrently.

Lock-free Coordinating

The literature has put out a number of progress requirements for lock-free data structures. The two most crucial requirements are wait-freedom [17] and non-blocking [20].

1. Regardless of the speed at which other processes execute, a data structure is said to be non-blocking if it ensures that a particular process can always finish its outstanding operation in a limited number of steps on its own (admits hunger).

2. Regardless of the speed at which other processes execute, a data structure is considered wait-free if it ensures that each process can always finish the tasks it has left undone in a certain number of steps (does not admit hunger).



The benefits of utilizing non-blocking algorithms (i.e., algorithms that meet the non-blocking property) include their immunity to process failures (no data corruption upon process failure), resilience to deadlocks and priority inversion, and lack of noticeable performance degradation from scheduling preemption, page faults, or cache misses. Since

non-blocking algorithms are frequently complex—every variation of a non-blocking queue is still a publishable result—they are still not widely used in real applications.

Although wait-free properties are desired, they impose too much complexity on implementation, limiting the ability for non-blocking to greatly increase concurrent application performance. Wait-free algorithms are seen to be less useful than non-blocking algorithms since they are frequently far more intricate and memory-intensive. Furthermore, collision avoidance strategies like exponential backoff can effectively manage famine.Several Resources

When synchronization is necessary, two or more resources are frequently involved. Take the issue of transferring funds between two bank accounts, for instance. The two accounts serve as the resources in this case, and the clerks—whether employed by the same bank or by separate ones—must coordinate their activities to avoid updating the same account concurrently.

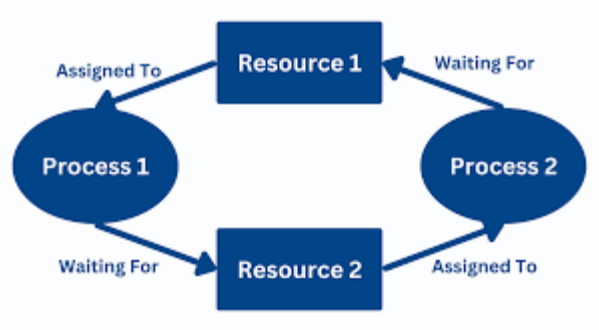
Deadlocks

Standstills

Assuming that there are thousands of clerks and millions of bank accounts—rather than just two—how would we "train the clerks" to avoid potential deadlocks? The sequel provides details on a fairly easy method that keeps such deadlocks from occurring. When there are several resources available, the concept of a deadlock is described as follows:

If every process in a set is waiting for an event that can only be caused by another process in the set, then the set of processes—or processors, or computers—is jammed.

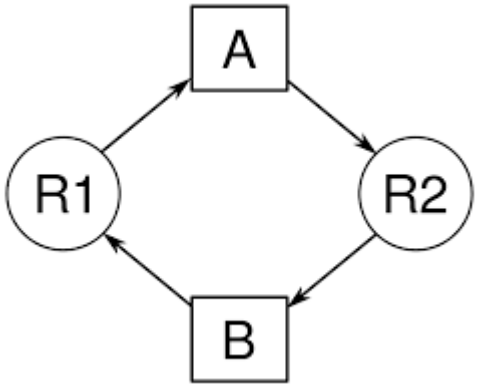
A currently held resource being released is typically the event described in the above definition.



Deadlock Prevention

Preventing Deadlocks

When correctly implemented, a number of straightforward design concepts can avoid deadlocks. The Method of Total Order. Imposing a complete ordering of all resources and mandating that each process request them in ascending order is the most effective preventive measure. More specifically, suppose that every resource has a number and that the numbers of the various resources are in some overall order. Deadlock cannot occur if processes request resources in a decreasing order of enumeration.



The complete order technique ensures that there isn't a loop in which multiple processes are waiting for a resource that is held by the process that comes after it. Deadlock is therefore impossible. This extremely basic method can now be applied to resolve the issue of money transfers between two bank accounts.

The two accounts are locked in ascending order, with the account numbers serving as the resource numbers. The absence of deadlock in this solution is ensured using the entire order technique. In the event that a resource required for the total order technique is unavailable, the process waits for the resource to become available, holding onto those resources it has already managed to lock.

We refer to these kinds of methods as hold and wait. In the next two methods, which are discussed below, the process may release all the resources it has already managed to lock and restart if a necessary resource is unavailable. We refer to these kinds of methods as "release and wait."

Deadlock Avoidance

The goal of the deadlock avoidance problem—dubbed the problem of Deadly Embrace by Dijkstra [10]—is to create an algorithm that can be used to determine whether resources can be efficiently allocated to requesting processes without creating the risk of a deadlock scenario through careful resource allocation.

Traditional Synchronization Issues

The list of significant synchronization issues is not exhaustive. Each of these issues has been covered in dozens of published papers. References and solutions to these issues are available in [34].

Barrier Time Alignment

Sometimes it makes sense to create algorithms that are phased so that no process may go on to the next one until all of the others have completed the previous phase and are prepared to move on to the next one together. Each stage of these algorithms usually rely on the outcomes of the one before it. At the conclusion of each phase, a barrier can be erected to encourage this kind of behavior.

Mutual Exclusion Locks

The exchange of conflict between processes is what resource allocation is all about. The challenge lies in resolving conflicts that arise when multiple processes attempt to utilize common resources. Stated differently, the distribution of common resources among competing activities. The mutual exclusion dilemma, in which there is only one resource accessible, is a specific instance of a general resource allocation problem. The guarantee of mutually exclusive access to a single shared resource in the presence of multiple competing processes is known as the mutual exclusion problem, or the difficulty of building a mutual exclusion lock [9].

FIFO, or first-in, first-out, A process that is already waiting for its turn to enter its critical section cannot begin its critical section earlier than any other process.

Deadlock freedom and mutual exclusion were the first two qualities that Dijkstra's original formulation of the problem required. These are the bare minimum standards that one could wish to set. It is assumed in the issue solving process that once a process begins executing its key part, it will always finish it, independent of what the other processes are doing. Among all the interprocess synchronization issues, the mutual exclusion problem has received the greatest research attention. This is a tricky problem that looks like it would be easy to tackle at first.

Multiple competing processes exist [9].

Multiple Resources

When synchronization is necessary, two or more resources are frequently involved. Take the issue of transferring funds between two bank accounts, for instance. The two accounts serve as the resources in this case, and the clerks—whether employed by the same bank or by separate ones—must coordinate their activities to avoid updating the same account concurrently.

In order to move money between two bank accounts, a clerk must first lock both of them, granting them exclusive access; then, after the money has been transferred, the locks on the two accounts must be released. The system may

stall if two clerks who need to transfer money between the same two accounts attempt to lock each account separately and in a different order, causing one clerk to lock an account and wait for the other.

References

1. Y. Afek, G. Stupp, and D. Touitou. Long lived adaptive splitter and applications. Distributed Computing, 30:67–86, 2002.

2. R. Alur and G. Taubenfeld.Results about fast mutual exclusion. In Proc. of the 13th IEEE Real-Time Systems Symposium, pages 12–21, 1992.

3. J. H. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. Proc. of the 14th international symposium on distributed computing. LNCS 1914:29–43, 2000.

4. T. E. Anderson. The performance of spin lock alternatives for shared-memory multiproces-sor. IEEE Trans. on Parallel and Distributed Systems, 1(1):6–16, 1990.

5. P.A. Bernstein and N. Goodman.
Timestamp-based algorithms for concurrency control
in distributed database systems. In Proc. of the International Conference on Very Large Databases, pages 285–300, 1980.

6. G. E. Blelloch, P. Cheng, and P. B. Gibbons.Room synchronization. In Proc. of the 13th Annual Symposium on Parallel Algorithms and Architectures, pages 122–133, 2001.

7. J.N. Burns and N.A. Lynch.Bounds on shared-memory for mutual exclusion. Information

and Computation, 107(2):171–184, 1993.

8. P.L. Courtois, F. Heyman, and D.L Parnas.Concurrent control with Readers and Writers. Communications of the ACM, 14(10):667–668, 1971.

9. E. W. Dijkstra. Solution of a problem in concurrent programming control. Communications of the ACM, 8(9):569, 1965.

10. E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, Programming Languages, pages 43–112. Academic Press, New York, 1968. Reprinted from: Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.

11. E. W. Dijkstra. Hierarchical ordering of sequential processes.ActaInformatica, 1:115–138, 1971.

12. K.P. Eswaran, J.N. Gary, A. Lorie, and I.L Traiger. The notion of consistency and predicate locks in database systems. Communications of the ACM, 19(11):624–633, 1976.

13. M.J. Fischer, N. A.Lynch, J.E. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. ACM TOPLAS, 11(1):90–114, 1989.

14. M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In Proc. 23rd ACM Symp. on Principles of Distributed Computing, pages 50–59, 2004.

15. G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. IEEE Computers, 28(6):69–69, 1990.

16. T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In Proc. 15th internationalsymp. on distributed computing, LNCS 2180:300–314, 2003.

17. M. P. Herlihy. Wait-free synchronization. TOPLAS, 13(1):124–149, 1991.

18. M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In Proc. of the 23rd International Conference on Distributed Computing Systems, pages 522–529, 2003.

19. M. P. Herlihy and J.E.B. Moss. Transactional memory: architectural support for lock-free

data structures. In Proc. of the 20th annual international symposium on Computer architecture, pages 289–300, 1993.

20. M. P. Herlihy and J. M. Wing.Linearizability: a correctness condition for concurrent objects. TOPLAS, 12(3):463–492, 1990.

21. H. Jordan. A special purpose architecture for finite element analysis. In Proc. of the Int. Conf. on Parallel Processing, pages 263–266, 1978.

22. Yuh-JzerJoung. Asynchronous group mutual exclusion. Distributed Computing, 13(4):189–206, 2000.

23. L. Lamport. A new solution of Dijkstra's concurrent programming problem.Communications of the ACM, 17(8):453–455, 1974.

24. L. Lamport. Concurrent reading and writing. Communications of the ACM, 20(11):806–811, 1977.

25. L. Lamport. A fast mutual exclusion algorithm. ACM Trans. on Computer Systems, 5(1):1–11, 1987.

26. L. Lamport. Concurrent reading and writing of clocks. ACM Trans. on Computer Systems, 8(4):305–310, 1990.

27. J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. on Computer Systems, 9(1):21–65, 1991.

28. M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In Proc. 15th ACM Symp.on Principles of Distributed Computing, pages 267–275, 1996.

29. S. S. Patil. Limitations of capabilities of Dijkstra's semaphore primitives for coordination among processes. In Project MAC computational structures group, MIT, memo 57, 1971.

30. M. O. Rabin. The choice coordination problem.ActaInformatica, 17:121–134, 1982.

31. N. Shavit and D. Touitou.Software transactional memory.In Proc. 14th ACM Symp.on

Principles of Distributed Computing, pages 204–213, 1995.

32. H. Sundell and P. Tsigas. Lock-free and practical deques using single-word compare-and-swap. In 8th International Conference on Principles of Distributed Systems, 2004.

33. G. Taubenfeld. The black-white bakery algorithm. In 18th international symposium on distributed computing, 2004. LNCS 3274:56–70, 2004.

34. G. Taubenfeld. Synchronization algorithms and concurrent programming. Pearson Education - Prentice-Hall, 2006. ISBN: 0131972596. www.faculty.idc.ac.il/gadi/book.htm.