Multiprocessors and Thread-Level Parallelism

1. Explain Symmetric Shared-Memory Architecture and How to reduce Cache coherence problem in Symmetric Shared Memory architectures:

Symmetric Shared Memory Architectures:

The Symmetric Shared Memory Architecture consists of several processors with a single physical memory shared by all processors through a shared bus which is shown below.

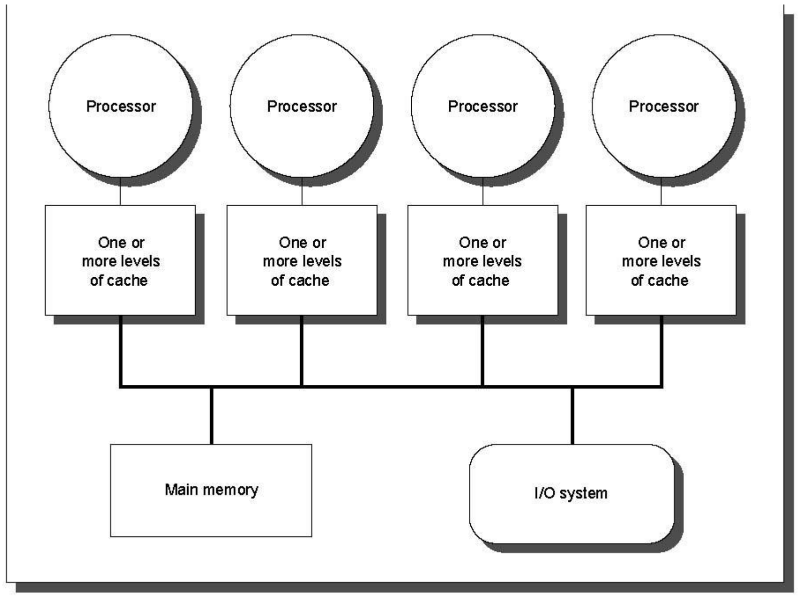
Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
I/O system

FIGURE 6.1 Basic structure of a centralized shared-memory multiprocessor. Multiple processor-cache subsystems share the same physical memory, typically connected by a bus. In larger designs, multiple buses, or even a switch may be used, but the key architectural property: uniform access time o all memory from all processors remains.

Small-scale shared-memory machines usually support the caching of both shared and private data. *Private data* is used by a single processor, while *shared data* is used by multiple processors, essentially providing communication among the processors through reads and writes of the shared data. When a private item is cached, its location is migrated to the cache, reducing the average access time as well as the memory bandwidth required. Since no other processor uses the data, the program behavior is identical to that in a uniprocessor.

Cache Coherence in Multiprocessors:

Introduction of caches caused a coherence problem for I/O operations, the same problem exists in the case of multiprocessors, because the view of memory held by two different processors is through their individual caches. Figure 6.7 illustrates the problem and shows how two different processors can have two different values for the same location. This difficulty s generally referred to as the *cache-coherence* problem.

Time	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A stores 0 into X	0	1	0

Figure 6.7 the cache-coherence problem for a single memory location (X), read and written by two processors (A and B).

We initially assume that neither cache contains the variable and that X has the value 1. We also assume a write-through cache; a write-back cache adds some additional but similar complications. After the value of X has been written by A, A's cache and the memory both contain the new value, but B's cache does not, and if B reads the value of X, it will receive 1! Informally, we could say that a memory system is coherent if any read of a data item returns the most recently written value of that data item. This simple definition contains two different aspects of memory system behavior, both of which are critical to writing correct shared-memory programs. The first aspect, called *coherence*, defines what values can be returned by a read. The second aspect, called *consistency*, determines when a written value will be returned by a read. Let's look at coherence first.
A memory system is coherent if

- A read by a processor, P, to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.
- A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.
- Writes to the same location are *serialized*: that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.

Coherence and consistency are complementary: Coherence defines the behavior of reads and writes to the same memory location, while consistency defines the behavior of reads and writes with respect to accesses to other memory locations.

Basic Schemes for Enforcing Coherence

Coherent caches provide migration, since a data item can be moved to a local cache and used there in a transparent fashion. This migration reduces both the latency to access a shared data item that is allocated remotely and the bandwidth demand on the shared memory.
Coherent caches also provide replication for shared data that is being simultaneously read, since the caches make a copy of the data item in the local cache. Replication reduces both latency of access and contention for a read shared data item.

The protocols to maintain coherence for multiple processors are called *cache-coherence protocols*. There are two classes of protocols, which use different techniques to track the sharing status, in use:
Directory based— The sharing status of a block of physical memory is kept in just one location, called the *directory;* we focus on this approach in section 6.5, when we discuss scalable shared-memory architecture.
Snooping— Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, and no centralized state is kept. The caches are usually on a shared-memory bus, and all cache controllers monitor or *snoop* on the bus to determine whether or not they have a copy of a block that is requested on the bus.

Snooping Protocols

The method which ensure that a processor has exclusive access to a data item before it writes that item. This style of protocol is called a *write invalidate protocol* because it invalidates other copies on a write. It is by far the most common protocol, both for snooping and for directory schemes. Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: all other cached copies of the item are invalidated.
Figure 6.8 shows an example of an invalidation protocol for a snooping bus with write-back caches in action To see how this protocol ensures coherence, consider a write followed by a read by another processor: Since the write requires exclusive access, any copy held by the reading processor must be invalidated (hence the protocol name). Thus, when the read occurs, it misses in the cache and is forced to fetch a new copy of the data. For a write, we require that the writing processor have exclusive access, preventing any other processor from being able to write simultaneously. If two processors do attempt to write the same data simultaneously, one of them wins the race, causing the other processor's copy to be invalidated. For the other processor to complete its write, it must obtain a new copy of the data, which must now contain the updated value. Therefore, this protocol enforces write serialization.

Processor activity	**Bus activity**	**Contents of CPU A's cache**	**Contents of CPU B's cache**	**Contents of memory location X**
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

Figure 6.8 An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches.
The alternative to an invalidate protocol is to update all the cached copies of a data item when that item is written. This type of protocol is called a *write update* or *write broadcast* protocol. Figure 6.8 shows an example of a write update protocol in operation. In the decade since these protocols were developed, invalidate has emerged as the winner for the vast majority of designs.

Processor activity	**Bus activity**	**Contents of CPU A's cache**	**Contents of CPU B's cache**	**Contents of memory location X**
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Write broadcast of X	1	1	1
CPU B reads X		1	1	1

Figure 6.9 An example of a write update or broadcast protocol working on a snooping bus for a single cache block (X) with write-back caches.

The performance differences between write update and write invalidate protocols arise from three characteristics:

- Multiple writes to the same word with no intervening reads require multiple write broadcasts in an update protocol, but only one initial invalidation in a write invalidate protocol.
- With multiword cache blocks, each word written in a cache block requires a write broadcast in an update protocol, although only the first write to any word in the block needs to generate an invalidate in an invalidation protocol. An invalidation protocol works on cache blocks, while an update protocol must work on individual words (or bytes, when bytes are written). It is possible to try to merge writes in a write broadcast scheme.
- The delay between writing a word in one processor and reading the written value in another processor is usually less in a write update scheme, since the written data are immediately updated in the reader's cache.

Basic Implementation Techniques

The serialization of access enforced by the bus also forces serialization of writes, since when two processors compete to write to the same location, one must obtain bus access before the other. The first processor to obtain bus access will cause the other processor's copy to be invalidated, causing writes to be strictly serialized. One implication of this scheme is that a write to a shared data item cannot complete until it obtains bus access.

For a write-back cache, however, the problem of finding the most recent data value is harder, since the most recent value of a data item can be in a cache rather than in memory. Happily, write-back caches can use the same snooping scheme both for caches misses and for writes: Each processor snoops every address placed on the bus. If a processor finds that it has a dirty copy of the requested cache block, it provides that cache block in response to the read request and causes the memory access to be aborted. Since write-back caches generate lower requirements for memory bandwidth, they are greatly preferable in a multiprocessor, despite the slight increase in complexity. Therefore, we focus on implementation with write-back caches. The normal cache tags can be used to implement the process of snooping, and the valid bit for each block makes invalidation easy to implement. Read misses, whether generated by an invalidation or by some other event, are also straightforward since they simply rely on the snooping capability. For writes we'd like to know whether any other copies of the block are cached, because, if there are no other cached copies, then the write need not be placed on the bus in a write-back cache. Not sending the write reduces both the time taken by the write and the required bandwidth.

2. Explain Distributed Shared-Memory Architectures.

There are several disadvantages in Symmetric Shared Memory architectures. First, compiler mechanisms for transparent software cache coherence are very limited. Second, without cache coherence, the multiprocessor loses the advantage of being able to fetch and use multiple words in a single cache block for close to the cost of fetching one word. Third, mechanisms for tolerating latency such as prefetch are more useful when they can fetch multiple words, such as a cache block, and where the fetched data remain coherent; we will examine this advantage in more detail later.
These disadvantages are magnified by the large latency of access to remote memory versus a local cache. For these reasons, cache coherence is an accepted requirement in small-scale multiprocessors. For larger-scale architectures, there are new challenges to extending the cache-coherent shared-memory model. Although the bus can certainly be replaced with a more scalable interconnection network and we could certainly distribute the memory so that the memory bandwidth could also be scaled, the lack of scalability of the snooping coherence scheme needs to be addressed is known as Distributed Shared Memory architecture. The first coherence protocol is known as a directory protocol. A *directory* keeps the state of every block that may be cached. Information in the directory includes which caches have copies of the block, whether it is dirty, and so on. To prevent the directory from becoming the bottleneck, directory entries can be distributed along with the memory, so that different directory accesses can go to different locations, just as different memory requests go to different memories. A distributed directory retains the characteristic that the sharing status of a block is always in a single known location. This property is what allows the coherence protocol to avoid broadcast. Figure 6.27 shows how our distributed-memory multiprocessor looks with the directories added to each node.

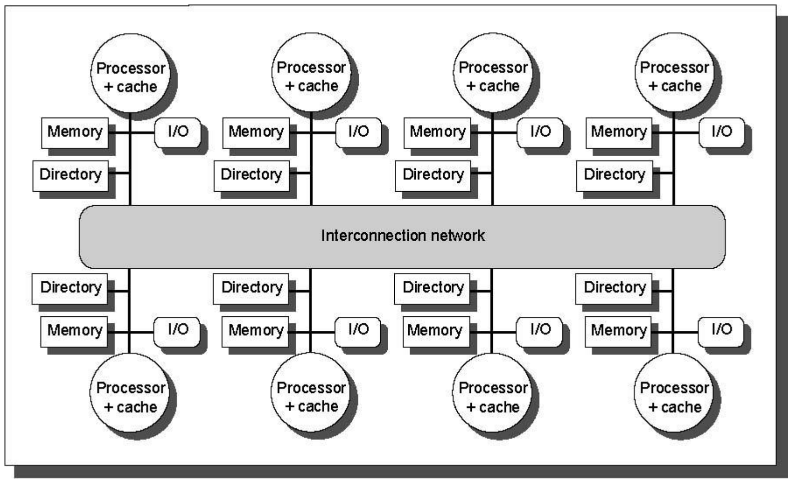

Figure 6.27 A directory is added to each node to implement cache coherence in a distributed-memory multiprocessor.

Directory-Based Cache-Coherence Protocols: The Basics

There are two primary operations that a directory protocol must implement: handling a read miss and handling a write to a shared, clean cache block. (Handling a write miss to a shared block is a simple combination of these two.) To implement these operations, a directory must track the state of each cache block. In a simple protocol, these states could be the following:
Shared—One or more processors have the block cached, and the value in memory is up to date (as well as in all the caches)
Uncached—No processor has a copy of the cache block
Exclusive—Exactly one processor has a copy of the cache block and it has written the block, so the memory copy is out of date. The processor is called the *owner* of the block.
In addition to tracking the state of each cache block, we must track the processors that have copies of the block when it is shared, since they will need to be invalidated on a write. The simplest way to do this is to keep a bit vector for each memory block. When the block is shared, each bit of the vector indicates whether the corresponding processor has a copy of that block. We can also use the bit vector to keep track of the owner of the block when the block is in the exclusive state. For efficiency reasons, we also track the state of each cache block at the individual caches.
A catalog of the message types that may be sent between the processors and the directories. Figure 6.28 shows the type of messages sent among nodes. The *local* node is the node where a request originates. The *home* node is the node where the memory location and the directory entry of an address reside. The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known. For example, the high-order bits may provide the node number, while the low-order bits provide the offset within the memory on that node. The local node may also be the home node. The directory must be accessed when the home node is the local node, since copies may exist in yet a third node, called a remote node.
A *remote* node is the node that has a copy of a cache block, whether exclusive (in which case it is the only copy) or shared. A remote node may be the same as either the local node or the home node. In such cases, the basic protocol does not change, but interprocessor messages may be replaced with intraprocessor messages.

Message type	**Source**	**Destination**	**Message contents**	**Function of this message**
Read miss	Local cache	Home directory	P, A	Processor P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Processor P has a write miss at address A; — request data and make P the exclusive owner.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.

Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write back	Remote cache	Home directory	A, D	Write back a data value for address A.

Figure 6.28 The possible messages sent among nodes to maintain coherence are shown with the source and destination node, the contents (where P=requesting processor number), A=requested address, and D=data contents), and the function of the message.

3. What is Synchronization? Explain various Hardware Primitives

Synchronization: Synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions. The efficient spin locks can be built using a simple hardware synchronization instruction and the coherence mechanism.

Basic Hardware Primitives: The key ability we require to implement synchronization in a multiprocessor is a set of hardware primitives with the ability to atomically read and modify a memory location. Without such a capability, the cost of building basic synchronization primitives will be too high and will increase as the processor count increases. There are a number of alternative formulations of the basic hardware primitives, all of which provide the ability to atomically read and modify a location, together with some way to tell if the read and write were performed atomically. These hardware primitives are the basic building blocks that are used to build a wide variety of user-level synchronization operations, including things such as locks and barriers.

One typical operation for building synchronization operations is the *atomic exchan*ge, which interchanges a value in a register for a value in memory. Use this to build a basic synchronization operation, assume that we want to build a simple lock where the value 0 is used to indicate that the lock is free and a 1 is used to indicate that the lock is unavailable. A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock. The value returned from the exchange instruction is 1 if some other processor had already claimed access and 0 otherwise. In the latter case, the value is also changed to be 1, preventing any competing exchange from also retrieving a 0.

There are a number of other atomic primitives that can be used to implement synchronization. They all have the key property that they read and update a memory value in such a manner that we can tell whether or not the two operations executed atomically. One operation, present in many older multiprocessors, is *test-and-set,* which tests a value and sets it if the value passes the test. For example, we could define an operation that tested for 0 and set the value to 1, which can be used in a fashion similar to how we used atomic exchange.

Another atomic synchronization primitive is *fetch-and-increment:* it returns the value of a memory location and atomically increments it. By using the value 0 to indicate that the synchronization variable is unclaimed, we can use fetch-and-increment, just as we used exchange. There are other uses of operations like fetch-and-increment.

Implementing Locks Using Coherence

We can use the coherence mechanisms of a multiprocessor to implement *spin locks:* locks that a processor continuously tries to acquire, spinning around a loop until it succeeds. Spin locks are used when a programmer expects the lock to be held for a very short amount of time and when she wants the process of locking to be low latency when the lock is available. Because spin locks tie up the processor, waiting in a loop for the lock to become free, they are inappropriate in some circumstances.

The simplest implementation, which we would use if there were no cache coherence, would keep the lock variables in memory. A processor could continually try to acquire the lock using an atomic operation, say exchange, and test whether the exchange returned the lock as free. To release the lock, the processor simply stores the value 0 to the lock. Here is the code sequence to lock a spin lock whose address is in R1 using an atomic exchange:

```
        DADDUI  R2,R0,#1
lockit:   EXCH                    R2,0(R1)          ;  atomic   exchange
        BNEZ                      R2,lockit         ;  already   locked?
```

If our multiprocessor supports cache coherence, we can cache the locks using the coherence mechanism to maintain the lock value coherently. Caching locks has two advantages. First, it allows an implementation where the process of "spinning" (trying to test and acquire the lock in a tight loop) could be done on a local cached copy rather than requiring a global memory access on each attempt to acquire the lock. The second advantage comes from the observation that there is often locality in lock accesses: that is, the processor that used the lock last will use it again in the near future. In such cases, the lock value may reside in the cache of that processor, greatly reducing the time to acquire the lock.

Synchronization Performance Challenges

Barrier Synchronization

One additional common synchronization operation in programs with parallel loops is a *barrier*. A barrier forces all processes to wait until all the processes reach the barrier and then releases all of the processes. A typical implementation of a barrier can be done with two spin locks: one used to protect a counter that tallies the processes arriving at the barrier and one used to hold the processes until the last process arrives at the barrier.

Synchronization Mechanisms for Larger-Scale Multiprocessors

Software Implementations

The major difficulty with our spin-lock implementation is the delay due to contention when many processes are spinning on the lock. One solution is to artificially delay processes when they fail to acquire the lock. The best performance is obtained by increasing the delay exponentially whenever the attempt to acquire the lock fails. Figure 6.41 shows how a spin lock with *exponential back-off* is implemented. Exponential back-off is a common technique for reducing contention in shared resources, including access to shared networks and buses. This implementation still attempts to preserve low latency when contention is small by not delaying the initial spin loop. The result is that if many processes are waiting, the back-off does not affect the processes on their first attempt to acquire the lock. We could also delay that process, but the result would be poorer performance when the lock was in use by only two processes and the first one happened to find it locked.

```
ADDUI               R3,R0,#1      ;R3 = initial delay
lockit:   LL        R2,0(R1)      ;load linked
          BNEZ      R2,lockit     ;not available-spin
          DADDUI R2,R2,#1         ;get locked value
          SC        R2,0(R1)      ;store conditional
          BNEZ      R2,gotit      ;branch if store succeeds
          DSLL      R3,R3,#1      ;increase delay by factor of 2
          PAUSE     R3            ;delays by value in R3
          J         lockit
gotit:    use data protected by lock
```

Figure 6.41 A spin lock with exponential back-off.

Another technique for implementing locks is to use queuing locks. Queuing locks work by constructing a queue of waiting processors; whenever a processor frees up the lock, it causes the next processor in the queue to attempt access. This eliminates contention for a lock when it is freed. We show how queuing locks operate in the next section using a hardware implementation, but software implementations using arrays can achieve most of the same benefits Before we look at hardware primitives,

Hardware Primitives: In this section we look at two hardware synchronization primitives. The first primitive deals with locks, while the second is useful for barriers and a number of other user-level operations that require counting or supplying distinct indices. In both cases we can create a hardware primitive where latency is essentially identical to our earlier version, but with much less serialization, leading to better scaling when there is contention.

The major problem with our original lock implementation is that it introduces a large amount of unneeded contention. For example, when the lock is released all processors generate both a read and a write miss, although at most one processor can successfully get the lock in the unlocked state. This sequence happens on each of the 20 lock/unlock sequences.

We can improve this situation by explicitly handing the lock from one waiting processor to the next. Rather than simply allowing all processors to compete every time the lock is released, we keep a list of the waiting processors and hand the lock to one explicitly, when its turn comes. This sort of mechanism has been called a *queuing lock*. Queuing locks can be implemented either in hardware, or in software using an array to keep track of the waiting processes.

4. What is Multithreading? How to exploiting TLP.

Multithreading: Exploiting Thread-Level Parallelism within a Processor

Multithreading allows multiple threads to share the functional units of a single processor in an

overlapping fashion. To permit this sharing, the processor must duplicate the independent state of each thread. For example, a separate copy of the register file, a separate PC, and a separate page table are required for each thread.
There are two main approaches to multithreading.
Fine-grained multithreading switches between threads on each instruction, causing the execution of multiples threads to be interleaved. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that time.
Coarse-grained multithreading was invented as an alternative to fine-grained multithreading. Coarse-grained multithreading switches threads only on costly stalls, such as level two cache misses. This change relieves the need to have thread-switching be essentially free and is much less likely to slow the processor down, since instructions from other threads will only be issued, when a thread encounters a costly stall.
Simultaneous Multithreading: Converting Thread-Level Parallelism into Instruction-Level Parallelism:
Simultaneous multithreading (SMT) is a variation on multithreading that uses the resources of a multiple issue, dynamically-scheduled processor to exploit TLP at the same time it exploits ILP. The key insight that motivates SMT is that modern multiple-issue processors often have more functional unit parallelism available than a single thread can effectively use. Furthermore, with register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability.

Issue Slots ➡

Superscalar Coarse MT Fine MT SMT

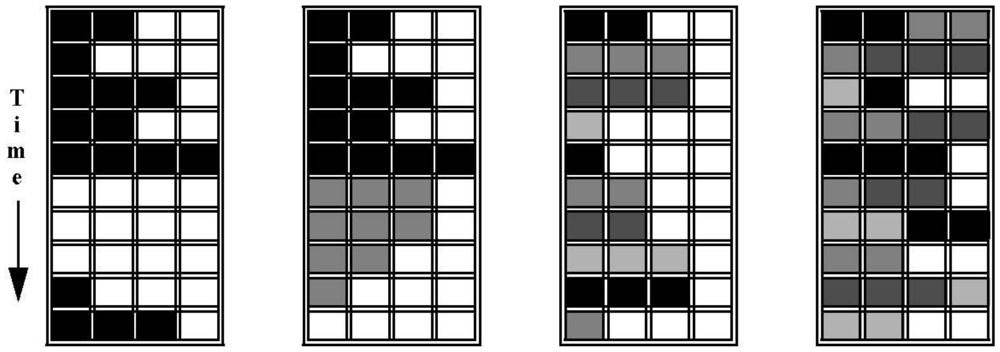

FIGURE 6.44 This illustration shows how these four different approaches use the issue slots of a superscalar processor. The horizontal dimension represents the instruction issue capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding issue slot is unused in that clock cycle. The shades of grey and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading support.

In the superscalar without multithreading support, the use of issue slots is limited by a lack of ILP.
In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor.

In the fine-grained case, the interleaving of threads eliminates fully empty slots. Because only one thread issues instructions in a given clock cycle.

In the SMT case, thread-level parallelism (TLP) and instruction-level parallelism (ILP) are exploited simultaneously; with multiple threads using the issue slots in a single clock cycle.
Figure 6.44 conceptually illustrates the differences in a processor's ability to exploit the resources of a superscalar for the following processor configurations:

n	a superscalar with no multithreading support,
n	a superscalar with coarse-grained multithreading,
n	a superscalar with fine-grained multithreading, and
n	a superscalar with simultaneous multithreading.

The above figure greatly simplifies the real operation of these processors it does illustrate the potential performance advantages of multithreading in general and SMT in particular.

Design Challenges in SMT processors

There are a variety of design challenges for an SMT processor, including:

- Dealing with a larger register file needed to hold multiple contexts,
- Maintaining low overhead on the clock cycle, particularly in critical steps such as instruction issue, where more candidate instructions need to be considered, and in instruction completion, where choosing what instructions to commit may be challenging, and
- Ensuring that the cache conflicts generated by the simultaneous execution of multiple threads do not cause significant performance degradation.

In viewing these problems, two observations are important. In many cases, the potential performance overhead due to multithreading is small, and simple choices work well enough. Second, the efficiency of current super-scalars is low enough that there is room for significant improvement, even at the cost of some overhead.

Dr. Dharavath Ramesh Hari Nandan
Assistant Professor & Inchage-Placments
Department of Computer Science & Engg.
Member IEEE, ACM, ISTE, IAENG
Indian Institute of Technology, Dhanbad
Email: drramesh@iitism.ac.in