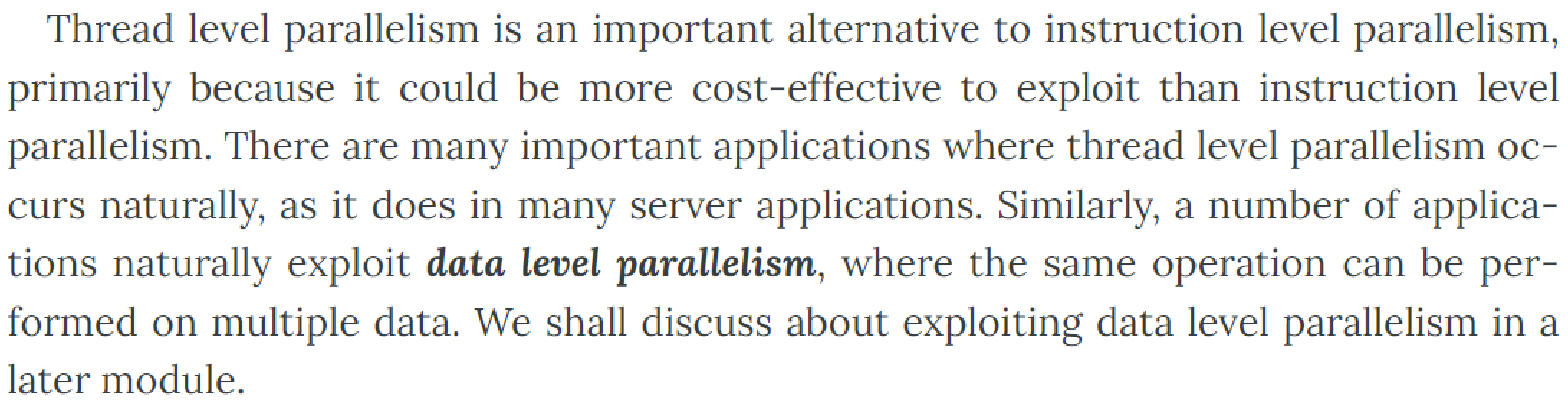
More on Parallelism

Limitations of ILP: Inspite of all the hardware and software techniques employed to exploit ILP, there is a limit to how much we can exploit ILP. First of all, there is a limitation with the hardware that we use. The number of virtual registers that we actually have is limited, not infinite, to do the renaming process. The branch predictors and jump predictors that we use may not be perfect. Similarly, we may not be able to resolve memory address disambiguities always. In short, we do not have an idealistic processor, limited only by true data dependences and without any control, WAR and WAW hazards.

Doubling issue rates above today's 3-6 instructions per clock, say to 6 to 12 instructions, probably requires a processor to issue 3 or 4 data memory accesses per cycle, resolve 2 or 3 branches per cycle, rename and access more than 20 registers per cycle, and fetch 12 to 24 instructions per cycle. The complexity of implementing these capabilities is likely to mean sacrifices in the maximum clock rate. For example, one of the widest issue processors is the Itanium 2, but it also has the slowest clock rate, despite the fact that it consumes the most power. Most techniques for increasing performance also increase the power consumption. Multiple issue processors techniques all are energy inefficient. Issuing multiple instructions incurs some overhead in logic that grows faster than the growth in issue rate. There is also a growing gap between the peak issue rates and sustained performance, which leads to increasing energy per unit of performance.

Exploiting other types of parallelism: The above discussion clearly shows that ILP can be quite limited or hard to exploit in some applications. More importantly, it may lead to increase in power consumption. Furthermore, there may be significant parallelism occurring naturally at a higher level in the application that cannot be exploited with the approaches used to exploit ILP. For example, an online transaction processing system has natural parallelism among the multiple queries and updates that are presented by requests. These queries and updates can be processed mostly in parallel, since they are largely independent of one another. This higher level parallelism is called ***thread level parallelism*** because it is logically structured as separate threads of execution. A *thread* is a separate process with its own instructions and data. A thread may represent a process that is part of a parallel program consisting of multiple processes, or it may represent an independent program on its own. Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute. Unlike instruction level parallelism, which exploits implicit parallel operations within a loop or straight-line code segment, thread level parallelism is explicitly represented by the use of multiple threads of execution that are inherently parallel.

Tread Level Parallelism



Thread level parallelism is an important alternative to instruction level parallelism, primarily because it could be more cost-effective to exploit than instruction level parallelism. There are many important applications where thread level parallelism occurs naturally, as it does in many server applications. Similarly, a number of applications naturally exploit ***data level parallelism***, where the same operation can be performed on multiple data. We shall discuss about exploiting data level parallelism in a later module.

Since ILP and TLP exploit two different types of parallel structure in a program, it is a natural option to combine these two types of parallelism. The datapath that has already been designed has a number of functional units remaining idle because of the insufficient ILP caused by stalls and dependences. This can be utilized to exploit TLP and thus make the functional units busy. There are predominantly two strategies for exploiting TLP along with ILP – ***Multithreading*** and its variants, viz., ***Simultaneous Multi Threading (SMT)*** and ***Chip Multi Processors (CMP)***. In the case of SMT, multiple threads share the same large processor which reduces under-utilization and does efficient resource allocation. In the case of CMPs, each thread executes on its own mini processor, which results in a simple design and low interference between threads. We will discuss about both these approaches.

Multithreading: Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion. In order to enable this, the processor duplicates the independent state of each thread – a separate copy of the register file, a separate PC, and a separate page table. The memory itself can be shared through the virtual memory mechanisms, which already support multiprogramming. In addition, the hardware must support the ability to change to a different thread relatively quickly; in particular, a thread switch should be much more efficient than a process switch, which typically requires hundreds to thousands of processor cycles.

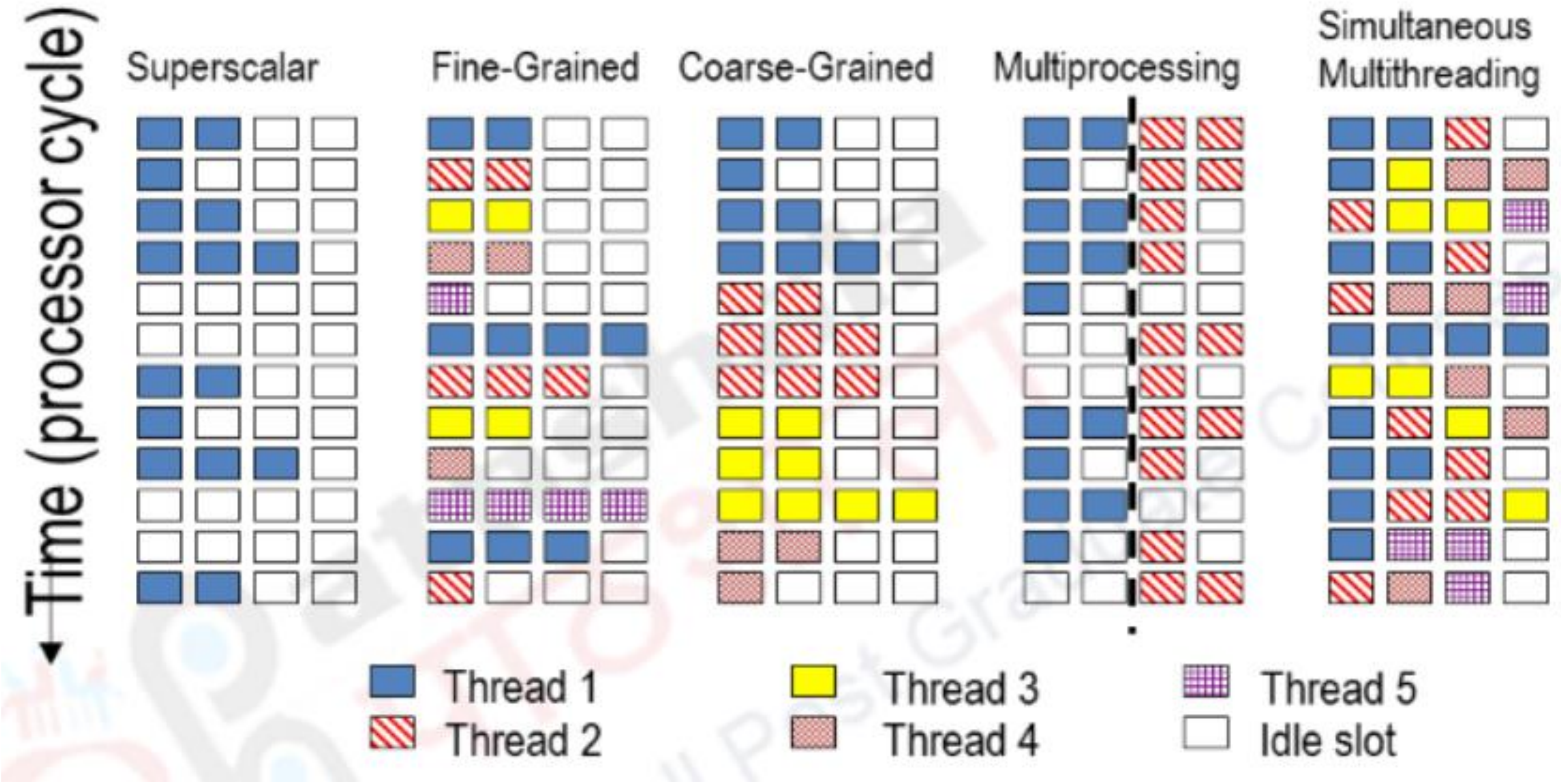
There are two main approaches to multithreading – Fine grained and Coarse grained. ***Fine-grained multithreading*** switches between threads on each instruction, causing the execution of multiple threads to be interleaved. This interleaving is normally done in a round-robin fashion, skipping any threads that are stalled at that time. In order to support this, the CPU must be able to switch threads on every clock cycle. The main advantage of fine-grained multithreading is that it can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls. But it slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

Coarse-grained multithreading switches threads only on costly stalls, such as level two cache misses. This allows some time for thread switching 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. Coarse-grained multithreading, however, is limited in its ability to overcome throughput losses, especially from shorter stalls. This limitation arises from the pipeline start-up costs of coarse-grain multithreading. Because a CPU with coarse-grained multithreading issues instructions from a single thread, when a stall occurs, the pipeline must be emptied or frozen and then fill in instructions from the new thread. Because of this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of high cost stalls, where pipeline refill is negligible compared to the stall time.

Simultaneous Multithreading: This is a variant on multithreading. When we only issue instructions from one thread, there may not be enough parallelism available and all the functional units may not be used. Instead, if we issue instructions from multiple threads in the same clock cycle, we will be able to better utilize the functional units. This is the concept of simultaneous multithreading. We try to use the resources of a multiple issue, dynamically scheduled superscalar to exploit TLP on top of ILP. The dynamically scheduled processor already has many HW mechanisms to support multithreading –

- a large set of virtual registers that can be used to hold the register sets of independent threads
- register renaming to provide unique register identifiers, so that instructions from multiple threads can be mixed in the data-path without confusing sources and destinations across threads and
- out-of-order completion that allows the threads to execute out of order, and get better utilization of the HW.

Thus, 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 will be handled by the dynamic scheduling capability. We need to add a renaming table per thread and keep separate PCs. The independent commitment of each thread can be supported by logically keeping a separate reorder buffer for each thread. Figure 24.1 shows the difference between the various techniques.



In the superscalar approach without multithreading support, the number of instructions issued per clock cycle is dependent on the ILP available. Additionally, a major stall, such as an instruction cache miss, can leave the entire processor idle. In the fine-grained case, the interleaving of threads eliminates fully empty slots. Because only one thread issues instructions in a given clock cycle, however, ILP limitations still lead to a significant number of idle slots within individual clock cycles. In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor. Although this reduces the number of completely idle clock cycles, within each clock cycle, the ILP limitations still lead to idle cycles. Furthermore, in a coarse-grained multithreaded processor, since thread switching only occurs when there is a stall and the new thread has a start-up period, there are likely to be some fully idle cycles. In the SMT case, TLP and ILP are exploited simultaneously, with multiple threads using the issue slots in a single clock cycle. Ideally, the issue slot usage is limited by imbalances in the resource needs and resource availability over multiple threads. In practice, other factors—including how many active threads are considered, finite limitations on buffers, the ability to fetch enough instructions from multiple threads, and practical limitations of what instruction combinations can issue from one thread and from multiple threads—can also restrict how many slots are used.

The other option that we need to discuss to exploit TLP and ILP is Chip Multi Processors (CMPs). Instead of looking at a powerful processor that might be a dynamically scheduled superscalar with support for speculation and also SMT, can we look at a simpler processor, but multiples of them? That is what CMPs stands for – several processors on a single chip. Each processor can individually support a thread of execution. Thus, with multiple processors, we have several threads of execution. These processors can have both shared and distributed memory architectures and they may be made up of both homogenous and heterogeneous processor types. Having several processors on the same chip reduces the wire delays. Since the processors are just replicated in most of the cases (homogenous), the very long design and verification times needed for modern complicated processors is avoided. The difference between an SMT processor and a CMP can be summarized as follows:

SMT versus CMT

- Simple Cores
  - – Moderate amount of parallelism
  - – Threads are running concurrently on different cores

SMT:

- Chip Multiprocessors integrate multiple processor cores on a single chip

- Pool of execution units (wide machine)
- Several Logical processors
  - – Copy of state for each of these logical processors
  - – Multiple threads run concurrently
  - – Better utilization and latency tolerance

- SMT
  - Pool of execution units (wide machine)
  - Several Logical processors
    - – Copy of state for each of these logical processors
    - – Multiple threads run concurrently
    - – Better utilization and latency tolerance
- CMP
  - Simple Cores
    - – Moderate amount of parallelism
    - – Threads are running concurrently on different cores
  - Chip Multiprocessors integrate multiple processor cores on a single chip
  - Reduces the hardware overhead.
  - Reduces power consumption.
  - CMP is an ideal platform to run multiprogrammed workloads or multithreaded applications. However, CMP architecture may lead to resource waste if an application cannot be effectively decomposed into threads or there is not enough TLP.