Thread 0	Thread 1
	load r1, A
	addi r1, r1, 3
load r1, A	
addi r1, r1, 1	
store r1, A	
	store r1, A

(a)

Thread 0	Thread 1
load r1, A	
addi r1, r1, 1	
store r1, A	
	load r1, A
	addi r1, rl, 3
	store r1, A

(b)

Thread 0	Thread 1
	load r1, A
	addi r1, r1, 3
	store r1, A
load r1, A	
addi r1, r1, 1	
store r1, A	

(c)

Thread 0	Thread 1
load r1, A	
addi r1, r1, 1	
	load r1, A
	addi r1, rl, 3
	store r1, A
store r1, A	

(d)

This figure shows four possible interleavings of the references made by two threads to a shared variable A, resulting in 3 different final values for A.

Figure 11.1: The Need for Synchronization.

Thread 0	Thread 1
fetchadd A, 1	fetchadd A, 3

(a)

Thread 0	Thread 1
spin:	spin:
cmpswp AL, 1	cmpswp AL, 1
bfail spin	bfail spin
load r1, A	load r1, A
addi r1, rl, 1	addi r1, rl, 3
store r1, A	store r1, A
store 0, AL	store 0, AL

(b)

Thread 0	Thread 1
spin:	spin:
ll r1, A	ll r1, A
addi r1, r1, 1	addi r1, r1, 3
stc r1, A	stc r1, A
bfail spin	bfail spin

(c)

Figure 11.2: Synchronization with (a) Fetch-and-Add, (b) Compare-and-Swap, and (c) Load-Linked/Store-Conditional.

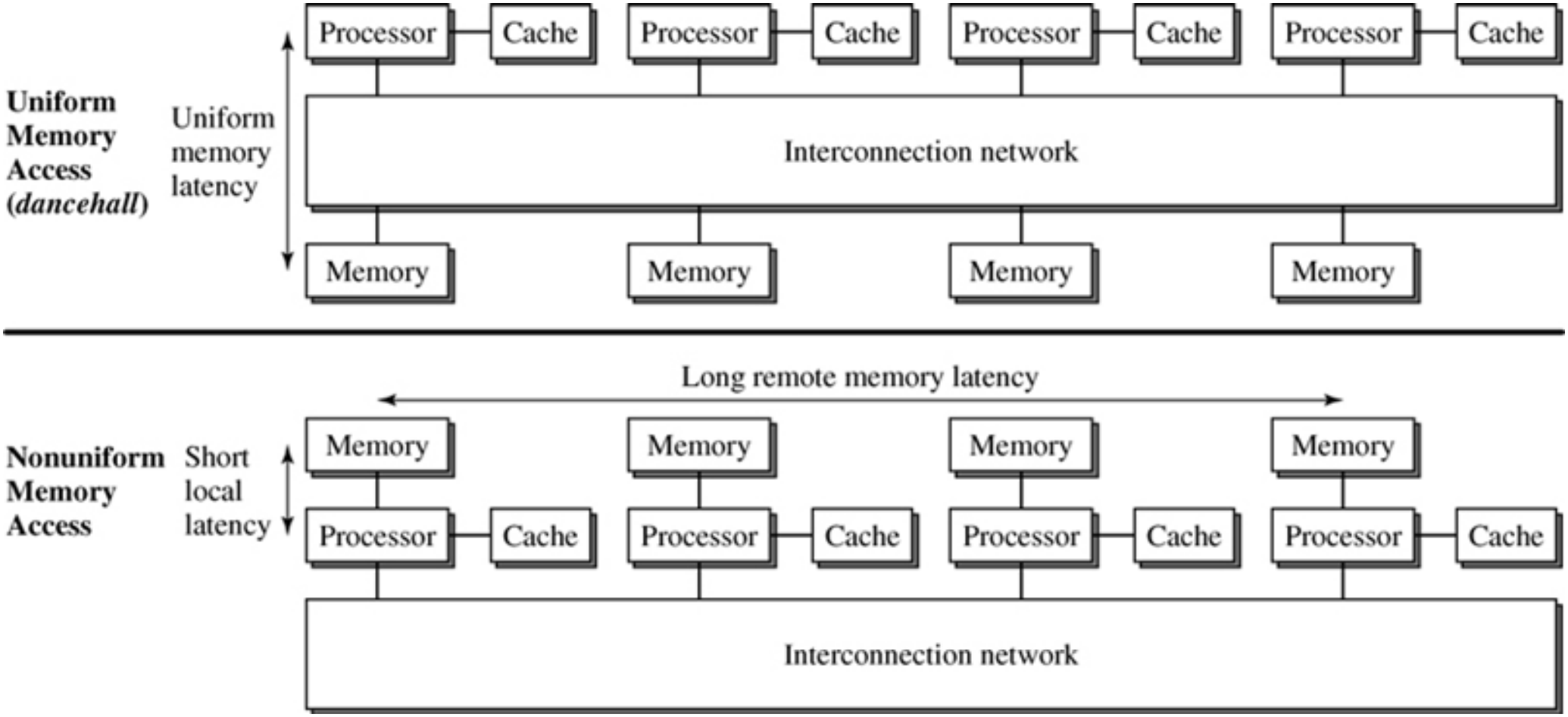
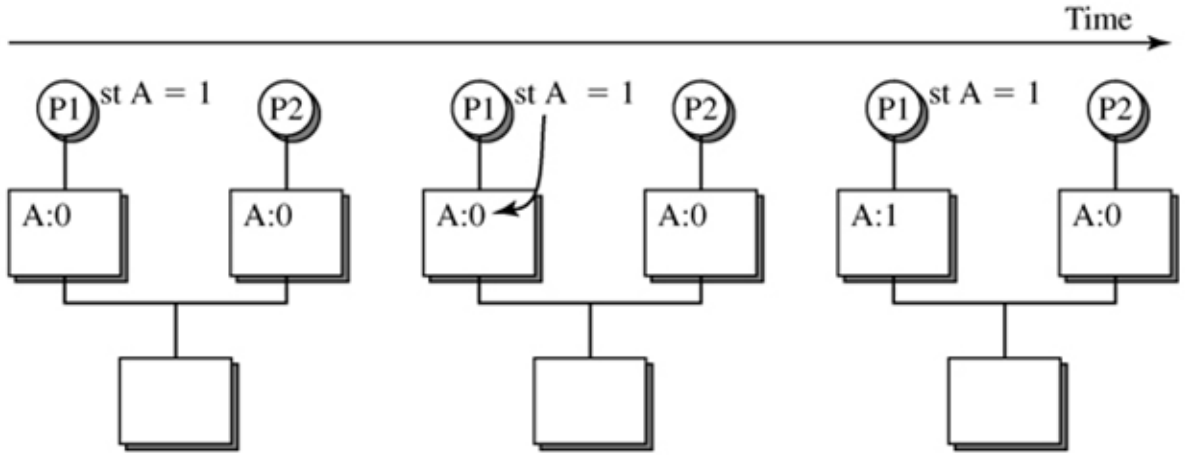
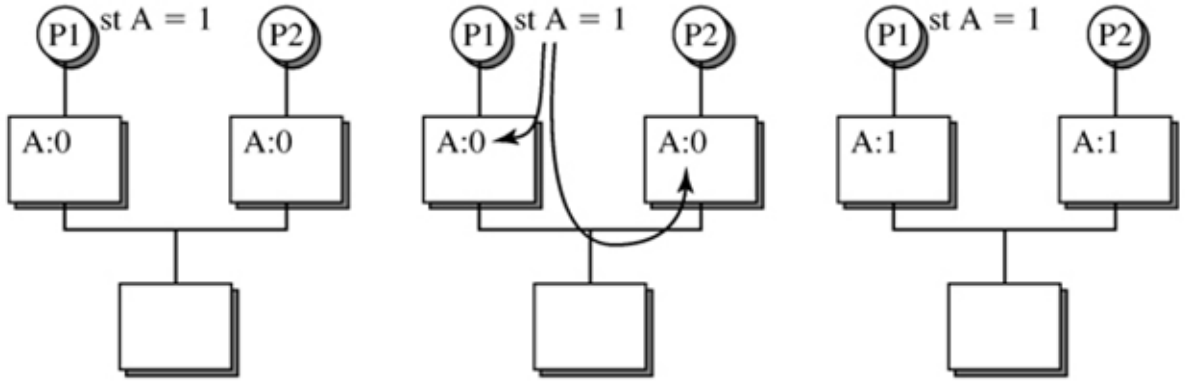


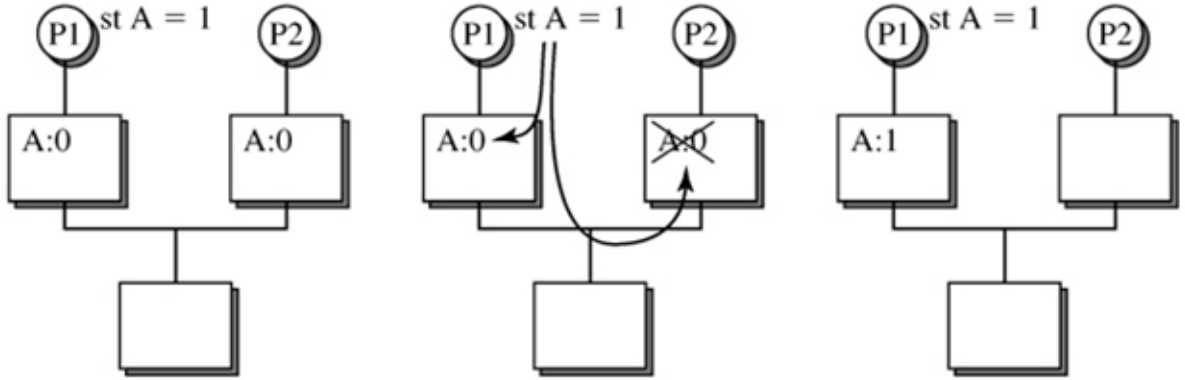
Figure 11.3: UMA versus NUMA Multiprocessor Architecture.



(a) No coherence protocol: stale copy of A at P2



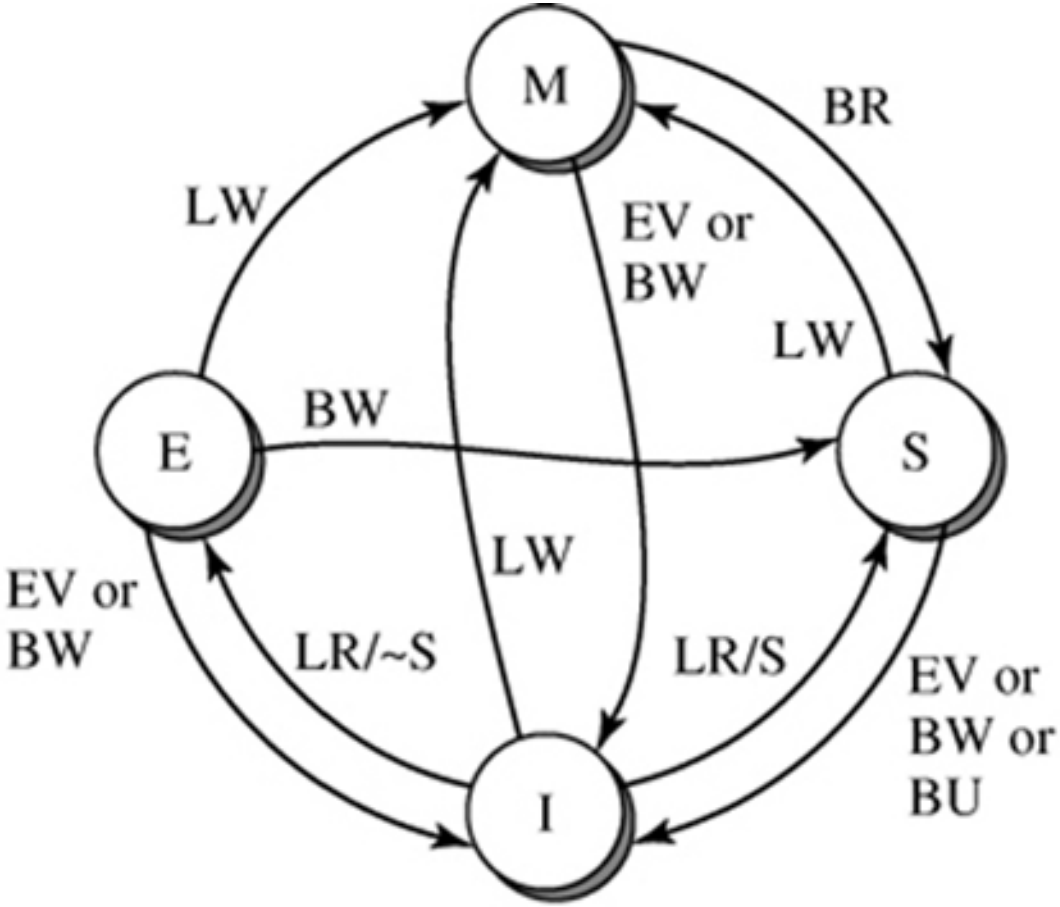
(b) Update protocol writes through to both copies of A



(c) Invalidate protocol eliminates stale remote copy

An update protocol updates all remote copies, while an invalidate protocol removes remote copies.

Figure 11.4: Update and Invalidate Protocols.



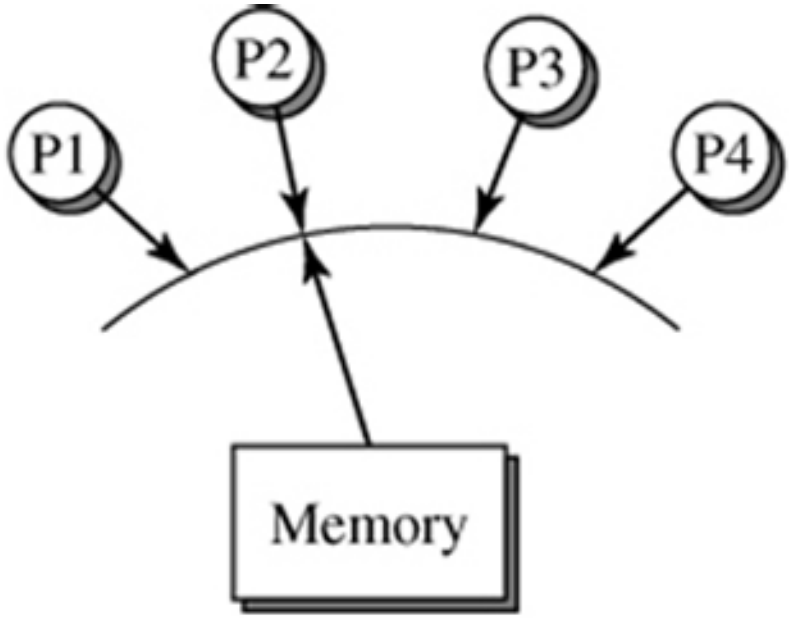
In response to local and bus events the coherence controller may need to change the local coherence state of a line, and may also need to fetch or supply the cache line data.

Figure 11.5: Sample MESI Cache Coherence Protocol.

```
Reorder  ↑  Proc0                          Proc1
load        |
before      |  st A=1                        st B=1
store       |  if (load B==0) {              if (load A==0) {
            |    ...critical section           ...critical section
            |  }                             }
```

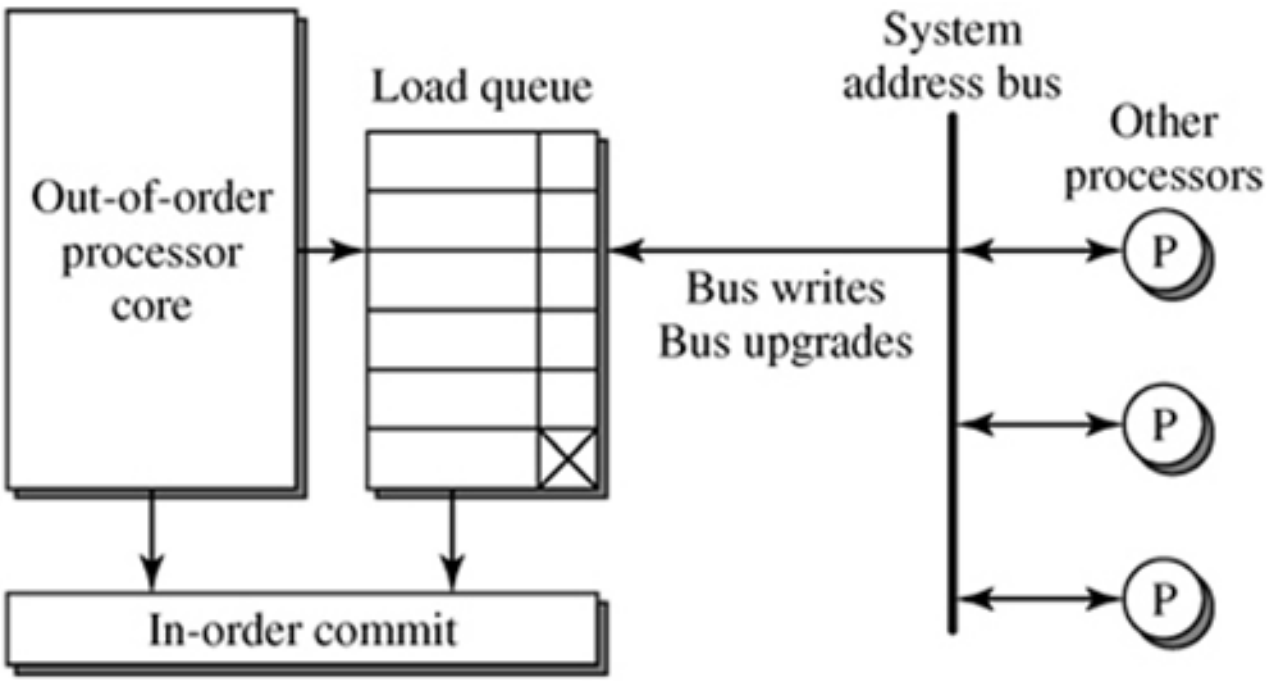
If either processor reorders the load and executes it before the store, both processors can enter the mutually exclusive critical section simultaneously.

Figure 11.6: Dekker's Algorithm for Mutual Exclusion.



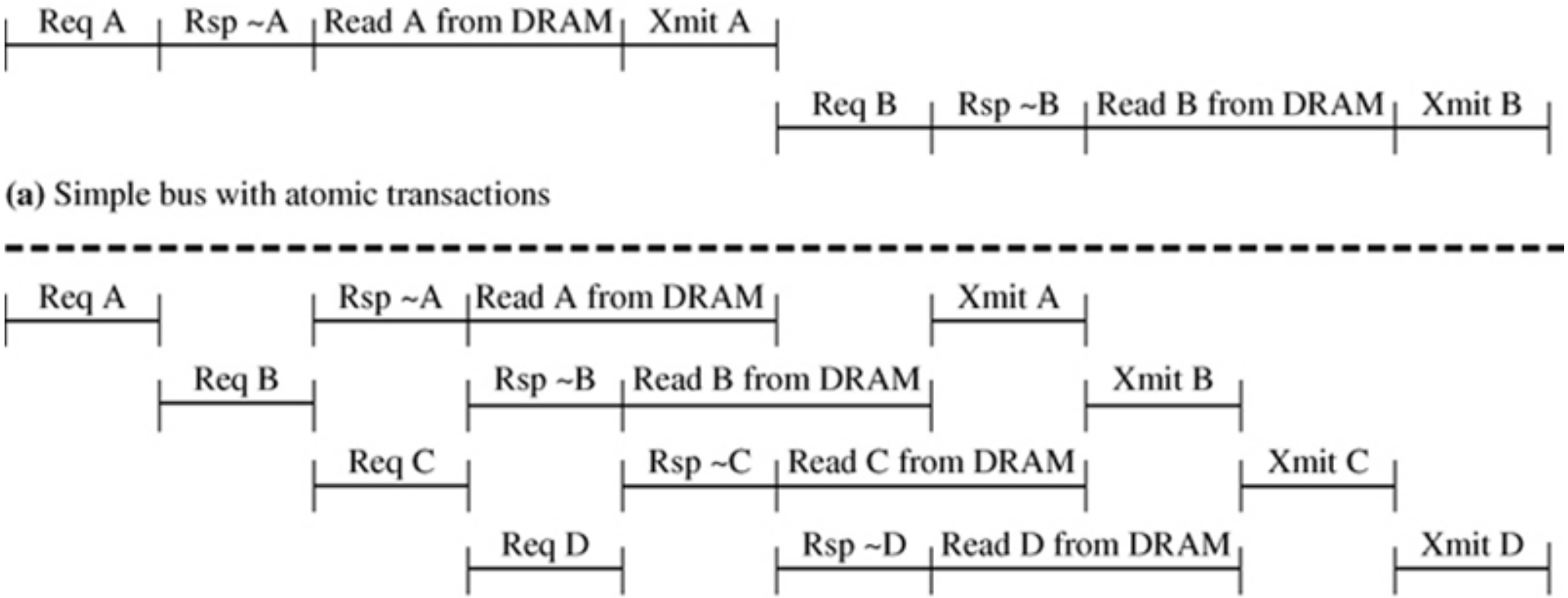
Each processor accesses memory in program order, and accesses from all processors are interleaved as if memory serviced requests from only one processor at a time.

Figure 11.7: Sequentially Consistent Memory Reference Ordering.
Source: Lamport, 1979.



Loads issue out of order, but loaded addresses are tracked in the load queue. Any remote stores that occur before the loads retire are snooped against the load queue. Address matches indicate a potential ordering violation and trigger refetch-based recovery when the load attempts to commit.

Figure 11.8: Read Set Tracking to Detect Consistency Violations.



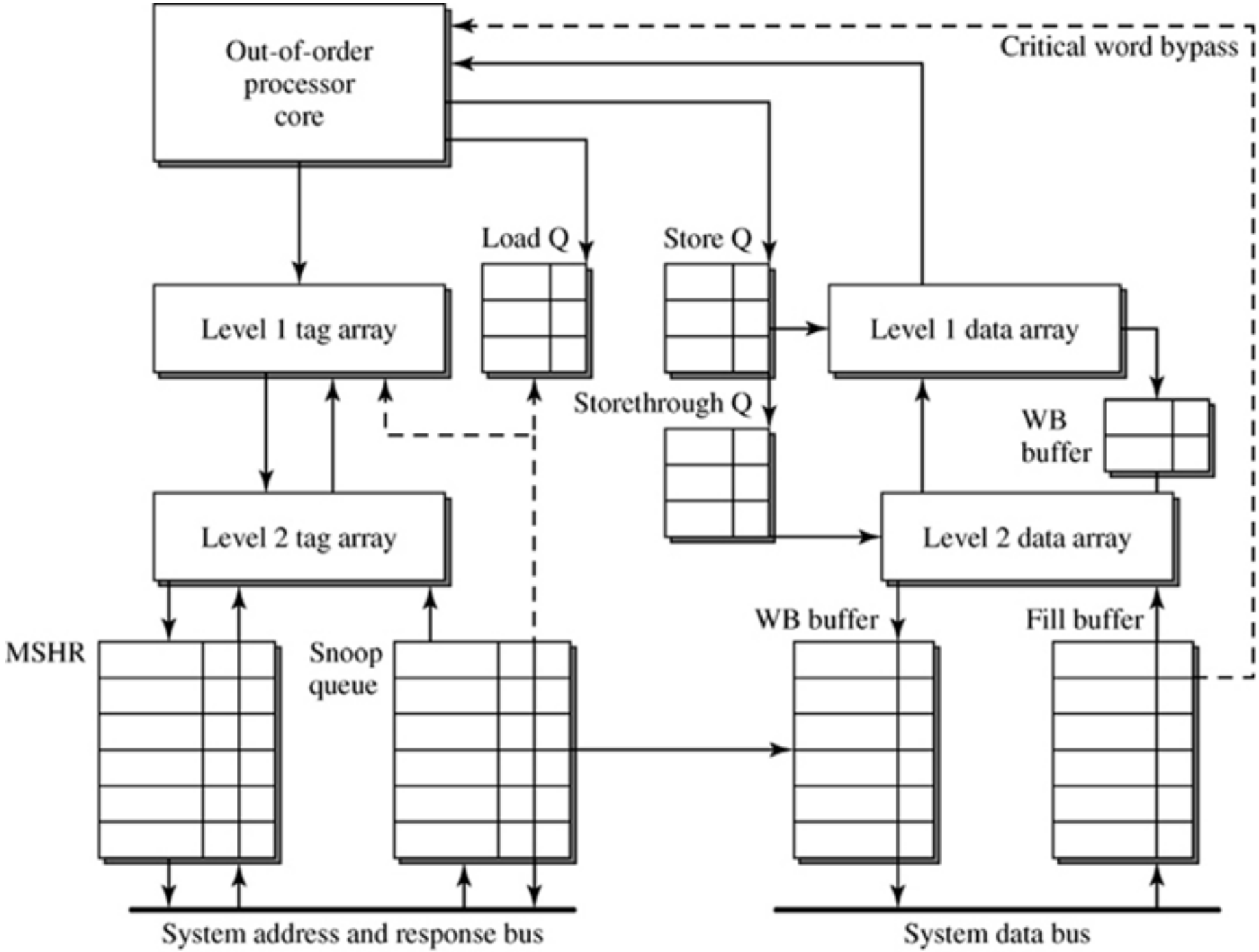
(a) Simple bus with atomic transactions

(b) Split-transaction bus with separate requests and responses

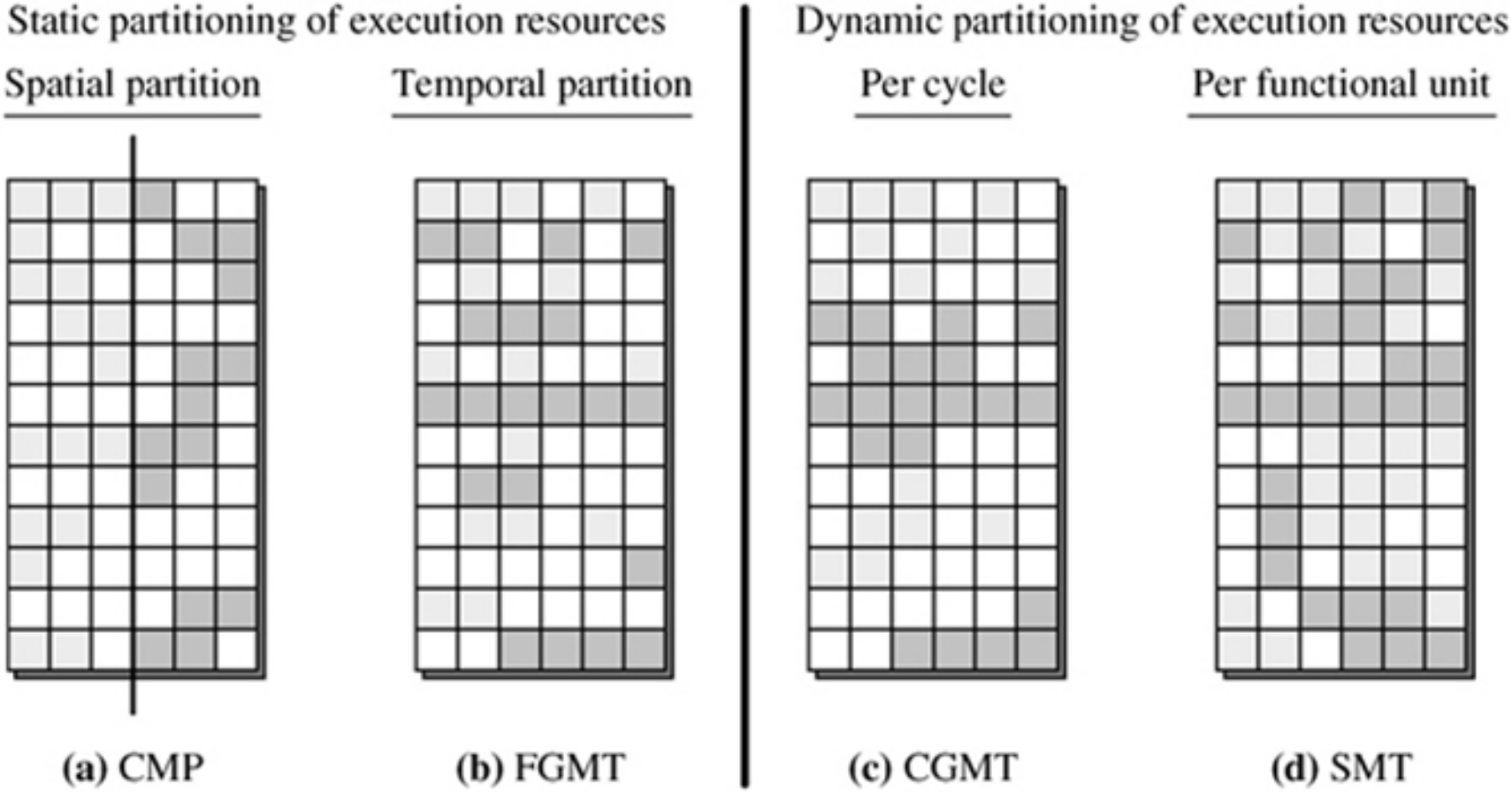
A split-transaction bus enables higher throughput by pipelining requests, responses, and data transmission.

Figure 11.9: Simple Versus Split-Transaction Busses.

Figure 11.10: Processor-Memory Interface.



A processor may communicate with memory through two levels of cache, a load queue, store queue, store-through queue (needed if L1 is write-through), MSHR (miss-status handling registers), snoop queue, fill buffers, and write-back buffers. Not shown is the complex control logic that coordinates all this activity.



Four possible alternatives are: chip multiprocessing (a), which statically partitions execution bandwidth; fine-grained multiprocessing (b), which executes a different thread in alternate cycles; coarse-grained multithreading (c), which switches threads to tolerate long-latency events; and simultaneous multithreading (d), which intermingles instructions from multiple threads. The CMP and FGMT approaches partition execution resources statically, either with a spatial partition by assigning a fixed number of resources to each processor, or with a temporal partition that time-multiplexes multiple threads onto the same set of resources. The CGMT and SMT approaches allow dynamic partitioning, with either a per-cycle temporal partition in the CGMT approach, or a per-functional unit partition in the SMT approach. The greatest flexibility and highest resource utilization and instruction throughput are achieved by the SMT approach.

Figure 11.11: Running Multiple Threads on One Chip.
Source: Tullsen et al., 1996.

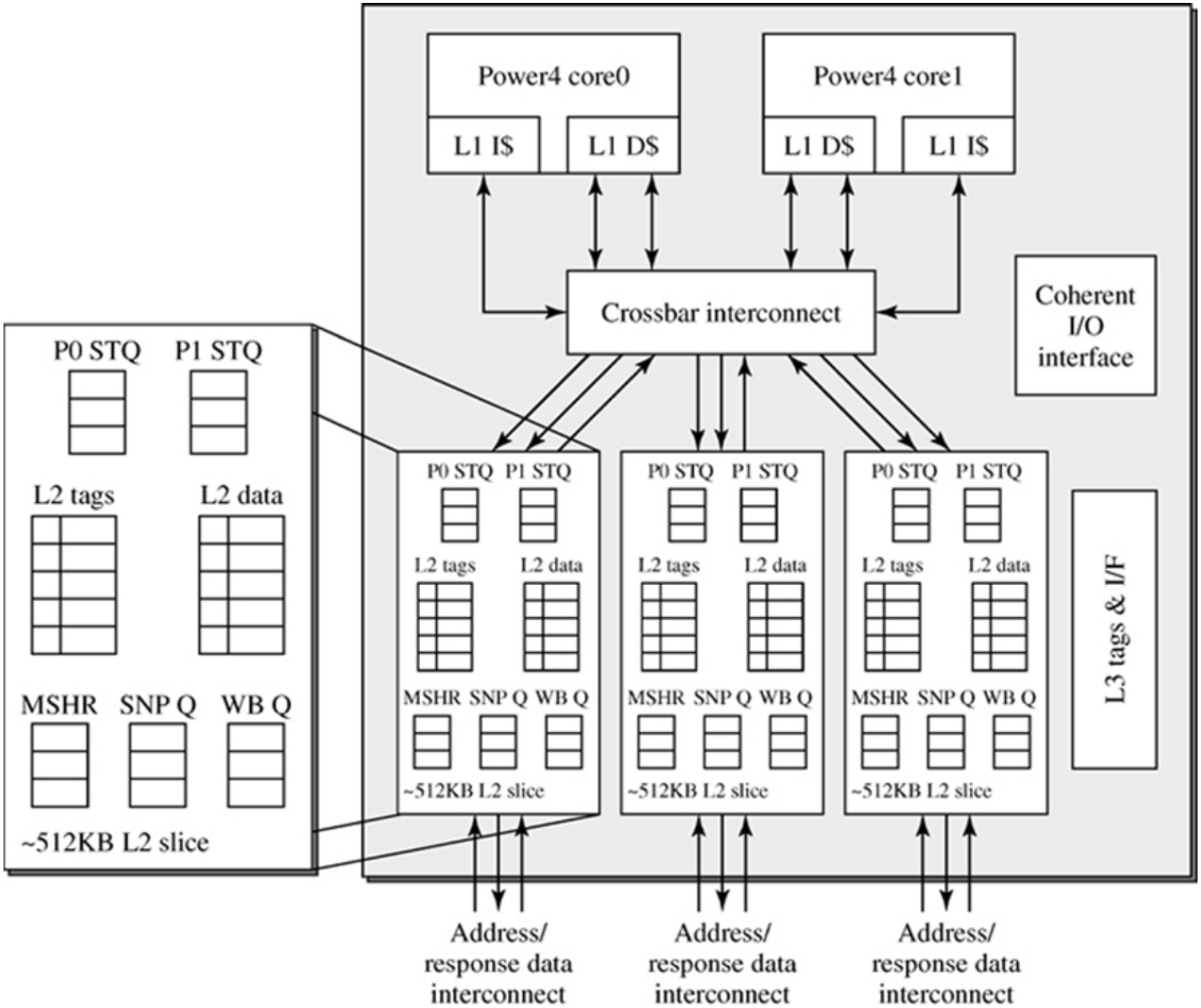


Figure 11.12: IBM POWER4: Example Chip Multiprocessor.
Source: Tendler et al., 2001.

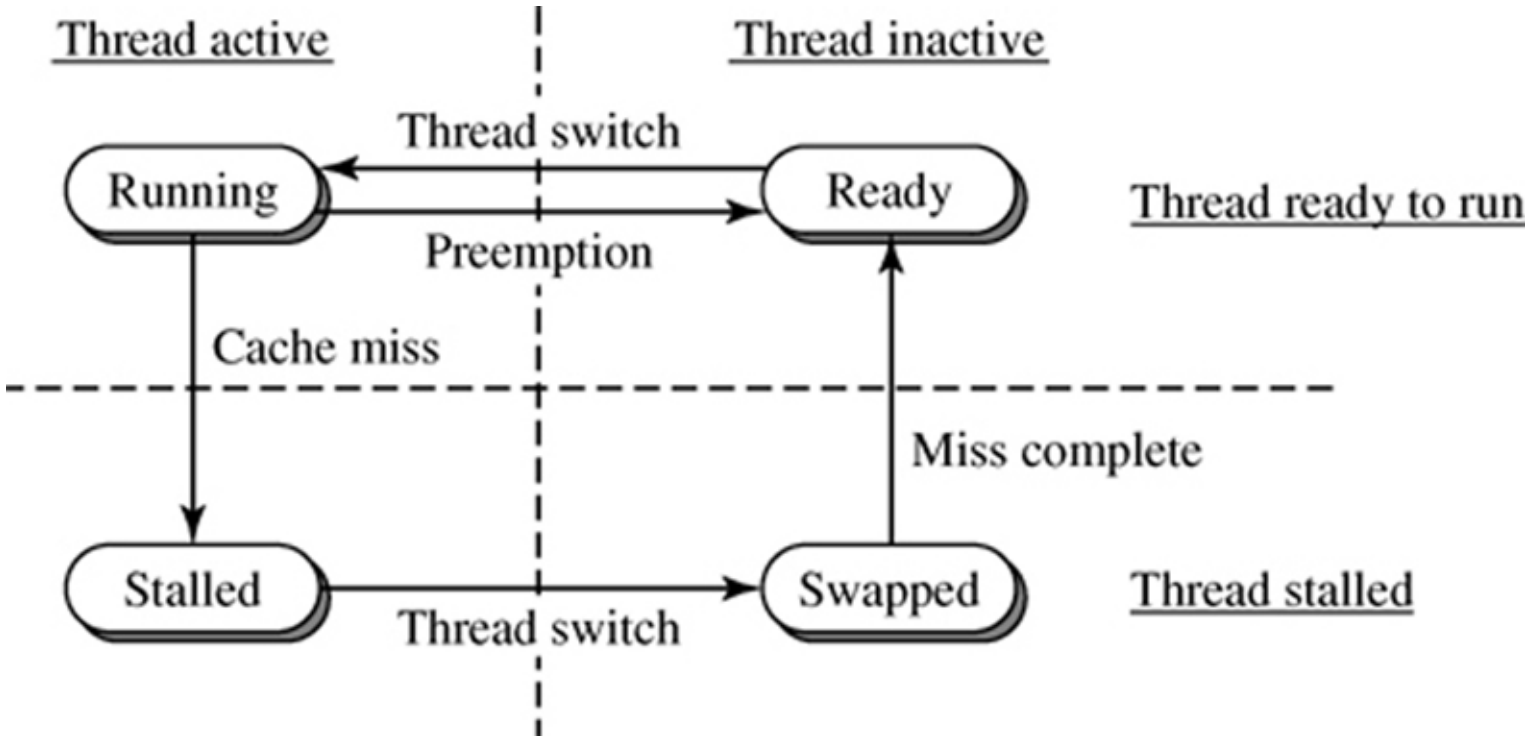


Figure 11.13: CGMT Thread Switch State Machine.

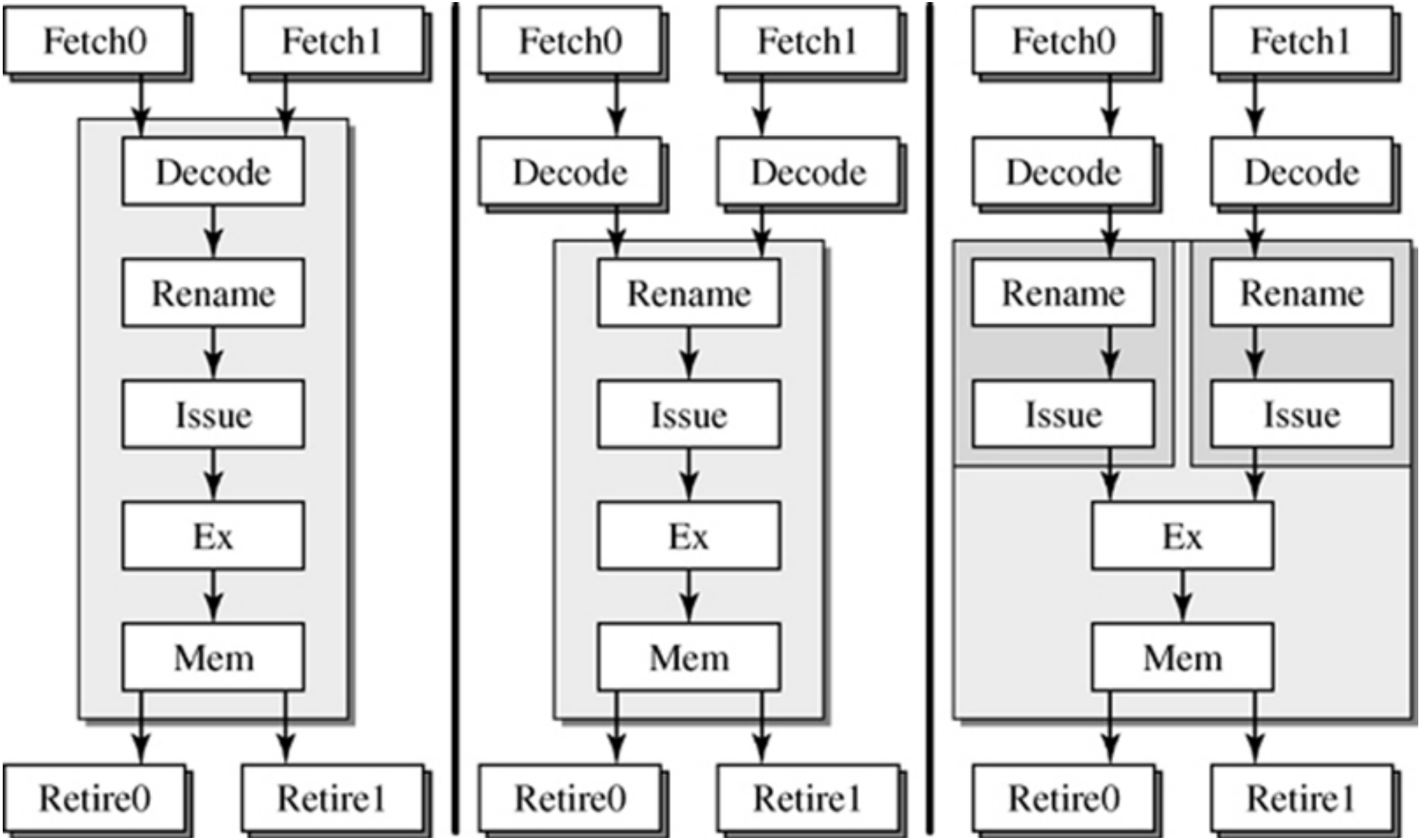
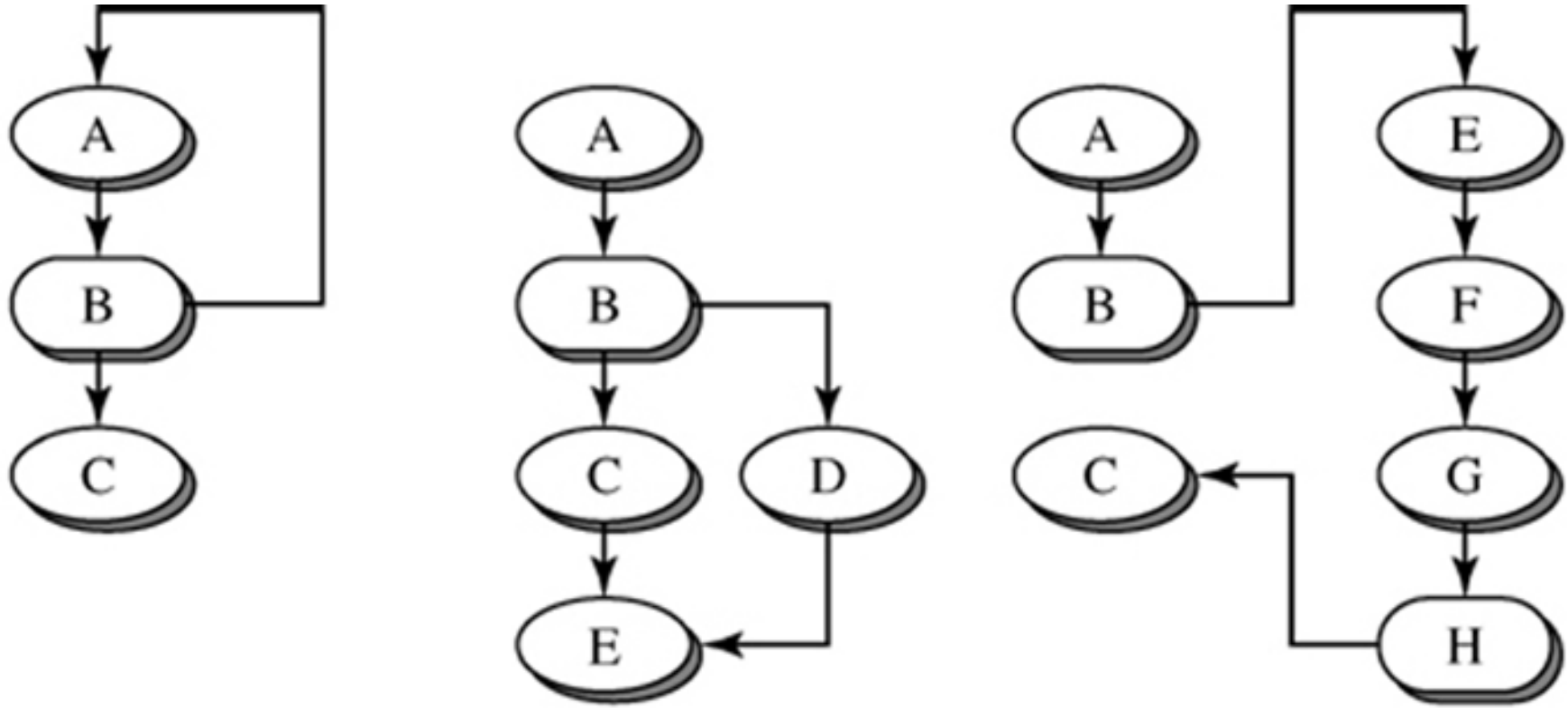


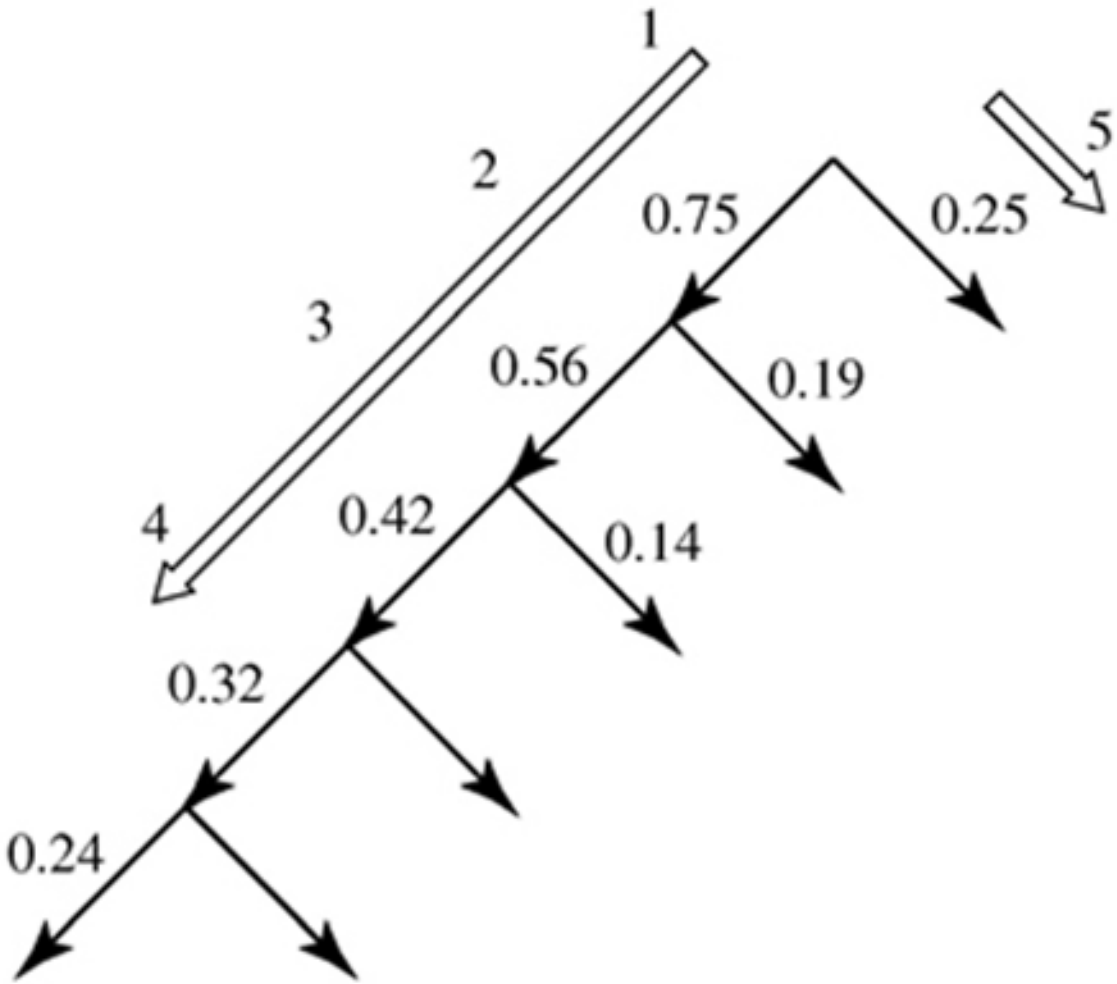
Figure 11.14: SMT Resource Sharing Alternatives.



(a) Loop-closing **(b)** Control-flow convergence **(c)** Call/return

There are multiple sources of control independence: in (a), block C eventually follows block B since the loop has a finite number of iterations; in (b) block E always follows B independent of which way the branch resolves; and in (c), block C eventually follows block B after the subroutine call to E completes.

Figure 11.15: Sources of Control Independence.



Assuming each branch is predicted with 75% accuracy, the cumulative branch prediction rate is shown; after fetching branch paths 1, 2, 3, and 4, the next-highest cumulative rate is along branch path 5, so it is fetched next.

Figure 11.16: Disjoint Eager Execution
Source: Uht and Sindagi, 1995.

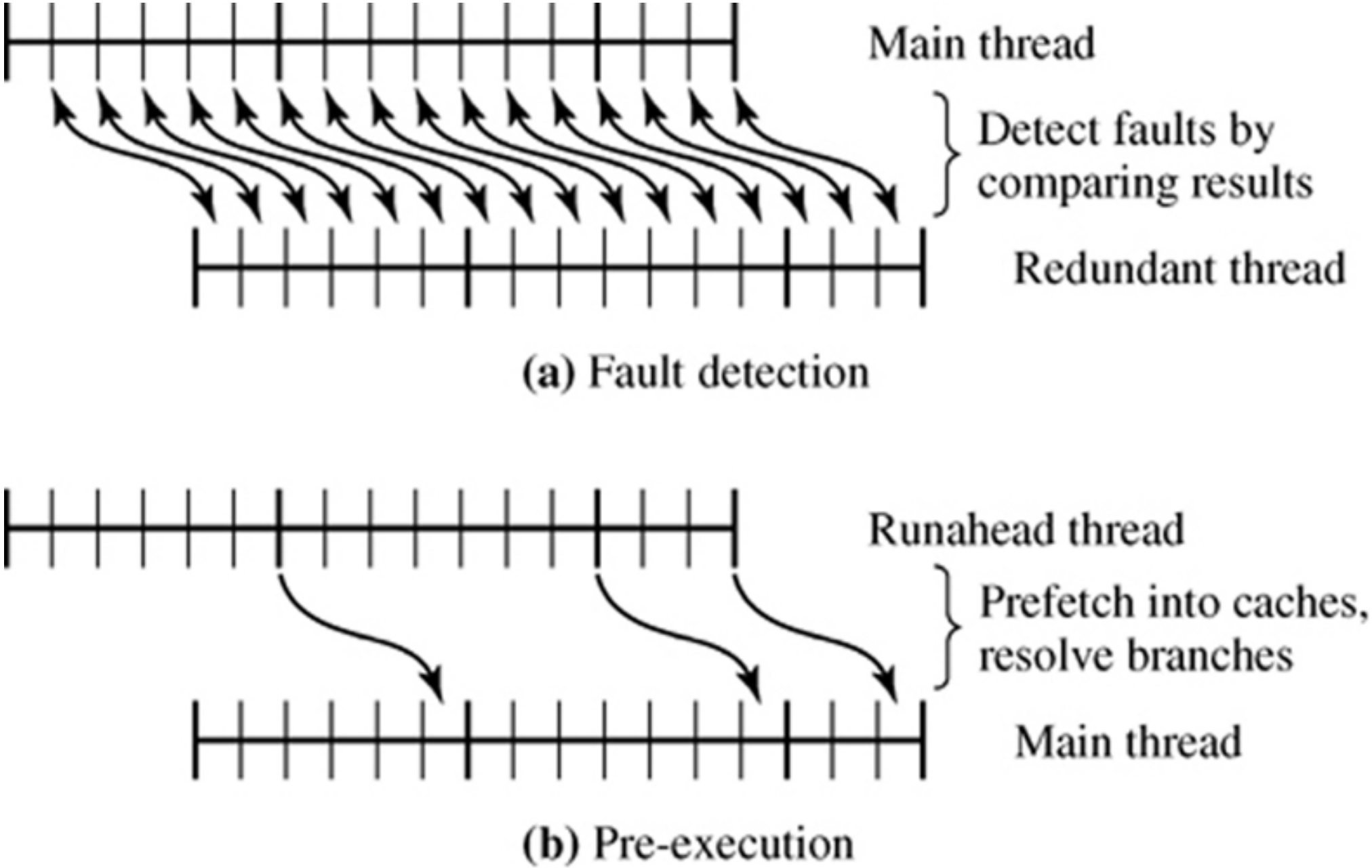


Figure 11.17: Executing the Same Thread.