# CSL862 Major

Answer all 24 questions 20/11/2012 Max Marks: 45

**Hybrid Transactional Memory**

1. Explain the following code generated by the hybrid transactional memory compiler and also explain how it preserves transactional semantics [3]

```
txn_begin handler-addr
if (!canHardwareRead(&X)) {
   txn_abort;
}
tmp= X;
if (!canHardwareWrite(&Y)) {
   txn_abort;
}
Y = tmp + 5;
txn_end;
```

txn_begin and txn_end are hardware instructions which specify the hardware transaction boundaries. This code is the code generate for a hardware transaction. Because software transactions co-exist with hardware transactions, concurrent reads and writes to shared variables in hardware transactions need to be protected against software transactions. This is done by maintaining an ownership record table (OREC_TABLE) which specifies the ownership status of a memory location. The compiler emits code in software transactions to atomically update the corresponding OREC (to read/write values) and the hardware transactions need to check those values using canHardwareRead and canHardwareWrite (emitted by the compiler). Full marks if all this is mentioned. 1 mark if do not mention “software transaction” and its interaction with hardware transaction.

Note that the hardware transaction does not set the ORECS – it only reads its values. If a software transaction modifies an OREC (before a conflicting hardware transaction commits), it will cause the hardware transaction to abort in hardware (because the OREC was a part of the hardware transaction’s read set and would now get invalidated).

2. Consider the “sequence locks “ described in the TxLinux paper. Why do you expect sequence locks to perform better than simple reader-writer locks? [2]

Just like read/write locks, sequence locks allow concurrent readers but enforce mutual exclusion on writers. For writers, they simply use a lock variable. Additionally they use a counter which is incremented before obtaining a lock and released after releasing the lock. Readers do not obtain any lock. They simply read the counter at the start and then read the counter at the end. Readers fail (rollback and retry) if they read an odd value in the counter as that means that a concurrent writer was updating the data. 0.5 marks if explain this.

Sequence locks are faster if there are frequent readers and occasional writers. Because a reader does not update any value (it only reads a counter twice), readers can update perfect scalability (no cache coherence misses) in the absence of writers. Read/write locks, on the other hand, are limited by the cache coherence traffic generated by the updates to the lock variable by the readers. Full marks if explain this.

3. From the TxLinux paper, look at the cx_exclusive(lock) function below:

```
void cx_exclusive(lock) {
   if (xgettxid)  xrestart(NEED_EXCLUSIVE);          //line 1
   while (1) {
      while (*lock != 1);    //spin
      disable_interrupts();
      if (xcas(lock, 1, 0))  break;                  //line 5
      enable_interrupts();
   }
}
```

   a. Explain the logic of "line 1". What is xgettxid and why is xrestart needed? [1]

cx_exclusive() obtains the lock in exclusive (mutex/pessimistic) mode. Hence, if I was already executing in a transaction I abort it and obtain the corresponding lock (pessimistic mode) as nested transactions are handled through flattening.

b. What are the semantics of xcas? How are they different from compare-and-swap? [1]

xcas consults the contention manager to decide who wins (e.g., between non-transactional and transactional thread), while CAS makes this decision arbitrarily. Xcas prevents problems like starvation of transactional thread.

4. In two of the papers/material we read, we found that it is not safe to write to a page table within a transaction. Why? [1]

Updates to PTEs can travel to hardware structures like TLBs and are very hard/expensive to undo on a transaction abort.

5. On TxLinux, explain using pseudo-code/diagram, how cxspinlocks can cause deadlocks in code that would otherwise not have had any deadlock (if using a simple spinlock). [2]

Thread 1 (non transactional):

cx_exclusive(cxspinb)

cx_exclusive(cxspina) // Assuming the contention policy always favours transactional thread, this statement will spin till Thread2 commits.

Thread2 (transactional):

txn_begin

cx_optimistic(cxspina)

cx_end(cxspina)

cx_optimistic(cxspinb) //spins on b

6. Explain “eager version management” vs. “lazy version management” and their tradeoffs in the context of “Operating Systems Transactions” paper. [2]

Eager: update data in place and maintain undo log. Costly to abort

Lazy: update private copies of data structure and update original copy on commit. Costly to commit

Tradeoff: Need to worry about deadlock issues in eager management for OS transactions. Deduct 0.5 marks if do not mention this.

7. In “OS Transactions”, both transactional and non-transactional threads co-exist. If the two threads (transactional and non-transactional) conflict, what are some contention management possibilities? For example, can the non-transactional thread be rolled back? Explain how starvation of transactional threads is prevented? [3]

A non-transactional thread cannot be rolled back, and hence the only option is to preempt it. This thread will be rescheduled when conflicting threads have committed. The other option is to rollback the transactional thread. This decision (between choosing the two options) is made by the conflict management policy. 1.5 marks for this

If a non-transactional thread is also non-preemptible, the transactional thread is always rolled back. To prevent starvation of the transactional thread in this case, the non-transactional thread can be put on the wait queue the next time it makes a system call to allow the transactional thread to commit. 1.5 marks for this

8. In Table 4 of “Operating System Transactions” paper, why does “NoTx” perform worse than “Static” column? Also, the mean overhead of the “Tx” column is 6.61x – is this acceptable performance? Why or why not? [2]

NoTx uses dynamic checks to determine if it is executing in transactional mode or not, while Static uses compiler support to compile separate versions for transactional and non-transactional execution and the runtime chooses one at kernel entry time. [1 mark]

The overhead of 6.61x on Tx column occurs only in the rare case when transactional operations are performed, so it is acceptable (though perhaps can be improved). [1 mark]

9. In Table 6 of “Operating System Transactions” paper, the caption states that “LDIF-TxOS provides the same crash consistency guarantees as BDB with more than double the write throughput”. Explain what this statement means, and why LDIF-TxOS could be performing better? [2]

BDB provides concurrency control mechanisms typical of any DB system. LDIF on the other hand is simply a flat file that writes at full speed with no concurrency control mechanism. However, if LDIF is used with OS Transactions, LDIF provides similar concurrency/crash control semantics as BDB. The results show that the overhead of implementing these transactional semantics are lower with TxOS implementation, than the Berkeley DB implementation.

10. In Figure 5 of “Operating Systems Transactions” paper, explain why TxOS performs better than “Linux-rename” at 8 processors? [1]

Because TxOS allows greater concurrency than coarser-grained locks used in Linux for filesystem operations.

11. Consider the experiment in Figure 2 and Figure 3 of the FlexSC paper. The direct cost dominates at high interrupt frequencies, and the indirect cost dominates at low interrupt frequencies. Explain why. [1]

As the interrupt frequency decreases, the effect of direct cost (trap) is amortized over a larger period. For indirect costs, this amortization happens much later (at a much lower frequency) because lower frequencies also imply more lost cache state on a syscall.

12. Explain the following statement:
    "The FlexSC system relies on the relatively fast cache-to-cache communication (in the order of 10s of cycles) available in modern multicore processors for performance of their architecture" [2]

    Because the syscall thread executes on a different core, one core (user thread) writes the syscall arguments to a shared page and another core (syscall thread) reads them. For this to remain efficient, fast cache-to-cache communication is required.

13. FlexSC uses a M-on-N threading package (M user threads executing on N kernel-visible threads) to harvest independent system calls by switching threads, in user-mode, whenever a thread invokes a system call. The “Scheduler Activations” paper suggested that this is a bad idea in general. For example, if one of the user thread busy waits for another thread that is currently switched out, it could degrade performance (or worse result in a deadlock on a strict priority system). Does it make sense to use scheduler activations with FlexSC? Why or why not? [4]

Some of the problems with M-on-N threading package are already taken care of by the fact that a system call never causes a user thread to block. This means that it is not possible for many user-level threads to have to wait behind a single blocked kernel level thread. (2 marks if specify this)

However there are still problems with interactions among user-level threads that could benefit from scheduler activations (like the one mentioned in the question). The problem is that if a process requests M user-level threads running on N kernel-level threads, and one of the N kernel-level threads gets swapped out (by the OS), it can create performance and deadlock-like problems. Essentially, it would be better for the user process to know the exact number of “virtual processors” available to it (instead of the kernel thread abstraction). This allows the user-level scheduler to swap-in/swap-out user-level threads accordingly. The user-level thread scheduler can then make the correct decisions and also provide the correct abstraction to its threads for them to decide on whether to busy-wait or not. (2 marks if specify this)

4 marks if specify both things

14. In FlexSC paper Figure 9, why is “flexsc” performing worse than “sync” when number of batched requests is 1 but shows improvement at 2 or more batched calls? [1]

Execution of a single syscall is expensive on FlexSC because it requires two context switches. Batching amortizes this overhead and also reduces the multiple trap overhead seen in sync.

15. Why is Figure 10 different from Figure 9? Why does FlexSC show improvements over sync even in this case at batching factor greater than 32? [1]

In Figure 10, the syscall thread executes on a remote core and thus involves an IPI ((interprocessor interrupt) which is more expensive. Amortizing IPI cost requires batching factor of >=32.

16. How do the FlexSC authors explain the disparity between the throughput improvement (94%) and the IPC improvement (71%) in the 4-core Apache throughput experiment? [2]

The difference is due to the added benefit of localizing the kernel execution with core specialization, which results in lowering the contention on a specific kernel semaphore.

17. In Barrelfish paper Figure 3, why is “SHM4” performing worse than “MSG8” but “SHM2” is performing better than “MSG8”? [2]

RPC implementation over shared memory involves maintaining a software queue with the producer appending to the queue and the consumer removing from that queue. For each update in MSG8, this involves, 2-4 (O(1)) cache coherence traffic. For SHM however, each update requires a cacheline invalidation and future reads of that value cause O(N) cache coherence traffic per update (N is the number of cores). Hence the extra cost of MSG mechanism is easily recovered at 4 or more cores (over SHM).

Only 0.5 marks if do not discuss how RPC mechanism is implemented and its consequences on cache coherence traffic.

18. Explain the following statement:
    "Message passing allows operations that might require communication to be *split-phase*". [1]

    There are operations where request and response can be decoupled. The requestor immediately continues after the request with the expectation that a reply will arrive in future. This reply can be obtained using polling or interrupt based mechanisms. Message passing allows such code organization.

19. Briefly explain why a "multikernel" is distinct from a "microkernel"? [1]

A microkernel also uses message-based communication between processes to achieve protection and isolation but remains a shared-memory multi-threaded system in the kernel. On the other hand, a multikernel (as described in the paper) uses a message passing kernel.

20. Briefly provide three reasons why a multikernel model is expected to be the fit OS architecture for future processors. [2]

1. Shared memory becomes a scalability bottleneck (cache coherence traffic)
2. Future cores may have heterogeneous cores (e.g., GPUs) which are a better fit for multikernel model (just spawn a kernel process on the heterogeneous core)
3. Multikernel can better adapt to the underlying hardware topology
4. Allows split-phase communication that can improve performance if hardware provides such primitives

21. In Barrelfish paper Figure 6 (TLB Shootdown experiment), explain why NUMA-aware multicast scales much better with the number of cores than either unicast or broadcast. [2]

NUMA-aware multicast makes optimum use of the hardware topology by first sending a message to one core of each processor and then letting that core send it to all its “neighbours” (i.e. cores belonging to the same processor). This reduces cache coherence traffic. Further, NUMA-awareness implies that the farthest core is sent the message first (to allow more overlap of communication). Deduct 0.5 marks if NUMA-awareness is not mentioned.

22. Why does Corey use a separate network stack for each core that requires network access? What are some implications of this to the external networked hosts and how are they handled? [2]

To minimize sharing (and contention). This implies that each core will have a separate IP address. Either the external hosts should treat each core as a separate server, or an ARP-based load balancer can be used.

23. Consider Corey paper Figure 8a. Why is “Dedicated” achieving better throughput than “Polling at less than 10 cores? Why do they perform similarly at 10 cores and higher? [2]

The “dedicated” line plots the performance when one core handles all network stack processing, while “polling” line plots the performance when all cores handle the network stack processing (thus causing sharing and contention). The latter is thus slower for less than 10 cores. At more than 10 cores, the network device bottlenecks.

24. Consider Corey paper Figure 12. The experiment involves checksumming a file on a webserver. The “locality mode” offers better throughput when filesize is between 256KB and 2048KB; otherwise both “locality” and “random” modes perform similarly. Explain why. [2]

In locality mode, the same core is likely to process a previously-seen file again allowing better temporal locality overall. The difference is seen only if the files are neither too small (because in this case, even random mode allows full caching of all files) and nor too large (because in this case, even the locality mode results in a lot of cache misses).