Patrick Santos (4465359)

Example questions and hints: Transactional Memory (TM)

Programming Problems

Q: The two code samples shown below are being run inside transactions on a multiprocessor system capable of using transactional memory. Assume a conflict is detected if one processor writes to a memory location that the other processor uses for a read or a write operation. If both pieces of code were executed at exactly the same time on both processors (assume there was a cache hit on both the instructions and data set so this was possible), would they be able to complete and commit in parallel? If not, is it possible for either of the two pieces of code to commit? Explain your reasoning..

(Assume all variables with the "global" prefix are shared, while any other variable is not, and will not cause conflits if written to during a transaction.)

Processor 1

```
transaction {
  // begin transaction
  globalSum = globalSum + localSum;
  globalCount = globalCount + 1;
  // end transaction
}
```

Processor 2

```
transaction {
  // begin transaction
  localTemp = globalSum;
  globalCount = globalCount – 1;
  // end transaction
}
```

A: Neither processor will be able to commit because there is a conflict of writes to globalCount. (this is designed to be a relatively simple problem that has little possiblity of different answers depending on assumptions made)

Q: Describe the changes that can be made to use transactional memory instead of locking with mutexes. Assume the target processor is capable of still using locks and barriers. Which lines can be included inside a transaction? Which locking and synchronization mechanisms can be removed?

```
// perform lengthy calculation in local space
if(  CPU_NUMBER == 0 ) {
  // processor 0 alone initializes the value to be read from during
  // calculation
  initialize( givens );
}
BARRIER();
X = performComplexCalculation( givens[CPU_NUMBER] );
LOCK(lock1);
globalSum = globalSum + X;
UNLOCK(lock1);
```

A: The barrier must remain since processors must be blocked until the value for calculations are ready, but the statment between the lock and unlock can be replaced with a transaction.

Algorithm Problems

Q: A hardware with a relatively naive transactional memory implementation in a multiprocessor system performs the follwing actions at the beginning, middle, and end of a transaction. For

simplicity, assume the processors have *no cache*, and all data needed for a transaction is stored on a dedicated buffer for transactions.

Beginning of transaction (speculate):

1. Set a register, called transaction-fail to 0.
2. Create local buffer to store current data register values, and memory locations to be updated at the end of the transaction.

During transaction:

1. Snoop on the bus. If any other processor issues a write to the memory locations used in this transaction, set transaction-fail to 1.

End of transaction (commit):

1. Check transaction-fail register.
   - If transaction-fail is 0: Commit values updated during the transaction to the main memory.
   - If transaction-fail is 1: Discard the local buffer. Branch to a subroutine to deal with the failed transaction. Restore the old data register values from the local buffer.

Does this transactional memory implementation suffer from deadlock, livelock, or starvation? For each one of deadlock, livelock, and starvation, explain why the system can or cannot suffer from that issue.

A: It cannot suffer from deadlock as no blocking is performed (this is a property of all transactional memory systems). It can suffer from livelock and / or starvation due to the way it only checks if the transaction failed at the end of the transaction. If processors attempt to place too much code and / or looping structions within the transactions, or retry immediately if transactions fail, the entire system can end up attempting transactions, failing, and retrying, causing livelock. Likewise, if one processor has a particularly long transaction, but other processors have short but frequent transactions affecting the same region of memory, the processor with the long transaction will be starved.

Computer Architecture Design Problems

Q: A hypothetical multiprocessor system with N (where N is greater than 1) processors has a local cache for each processor, and implements a transactional memory system. Assume this cache has only one level, and it is large enough to contain the entire working set for a transaction. You wish to make use of the MESI cache coherence protocol which is implemented on this cache to check if a transaction has failed. Which state transitions should be monitored to check if there was a conflict and the transaction must be aborted?
(e.g. M to I?)
Hint: Abort on any state transition that signals that another processor may have modified the value of a variable used in the transaction. (See section VII of report: Sun signals an abort on any transition to invalid.)

Q: A hardware transactional memory implementation stores speculative data (the data generated by writes to variables within a transaction) in a dedicated buffer. This buffer is used on a commit, where all the data that was changed during a transaction is stored to the main memory. If a transaction fails, the buffer is simply discarded. (do not allocate any space in the buffer for use on an abort) AT LEAST how large does this buffer have to be to store the data from the transaction shown in the code sample below?
Assume the following:

- array and the value used for the loop counter are 32-bit integers
- CPU_NUMBER is a value stored locally for the processor.
- All variables except globalArray are local, but all variables are stored in the same (shared)

memory.

- no room has to be allocated for user registers, program counter, or any other status flags and registers not mentioned inside the block of code below

```
int i = 0;
transaction {
  // begin transaction
  for( i = 0; i < 32; i = i + 1) {
    // add the 32 contributions of this processor to the global array
    gloabalArray[i + CPU_NUMBER*32] = localArray[i];
  }
  // end transaction
}
```

A: Note that globalArray[X to X + 32], as well as the loop variable i are written to during the transaction. Therefore, you need at least 33 x 32 bits = 1056 bits to store the speculative data. The values of localArray are never changed during the transaction, so no storage for localArray was required.

General Questions

Q: What is the difference between lazy and eager conflict detection in TM?
A: Lazy conflict detection waits until the end of a transaction to check for conflicts, while eager detection checks for conflicts continuously or at each read and write (see section V of report).

Q: What can limit the size of a transaction in an hardware transactional memory (HTM) system?
A: Buffers and / or cache size can limit the transaction size (see section V of report).

Q: How can a software TM implementation suffer from longer latency compared to hardware TM implementations?
A: Rather than copying from a buffer local to the processor, the transaction data must be copied from some region in main memory if a commit or abort is performed (see section V of report).

Q: Which of the following must be returned to the previous state in a TM for an abort, assuming the system should be livelock free? (you may select more than one item)

1. Program counter
2. Local registers
3. Modified variables

A: 2 and 3: restoring program counter would make the transaction happen over and over again, potentially causing livelock if many processors are doing this at once

Q: Why is TM deadlock free as a mechanism to ensure mutual exclusion?
A: TM does not block (see section VI of report).