15-418/618, Spring 2018

Exam 2 Practice SOLUTIONS

April, 2018

Warm Up: Miscellaneous Short Problems

Problem 1. (21 points):

A. (3 pts) Consider the following code where two threads are running on different cores of a two-core processor that features relaxed memory consistency (assume all variables start out with value 0). The system allows write-after-write and read-after-write memory operations to be reordered. (You should assume that the compiler itself performs **no instruction reordering**).

```
  Thread 1                    Thread 2
=============================================

  A = 1                  while (flag == 0) {}
  flag = 1               print A
  print A
```

Is it possible for Thread 1 to print the value 0 on this system? Is it possible for Thread 2 print the value 0? In both cases, please justify your answer.

Solution: It is ***not possible*** *for thread 1 to print 0, since the statement* `print A` *has a program order dependency on the proceeding statement* `A=1`*. Even if a system features relaxed memory consistency, it will not ever violate program order. It* ***is possible*** *for thread 2 to print 0. Since the system relaxes write after write ordering, thread 2 may observe the write to* `flag` *prior to observing the write to* `A`*. (Relaxed consistency troubled students on the midterm so as advertised we brought it back up on Exam 2 again.)*

B. (3 pts) Your job is to implement an iPhone app that continuously processes images from the camera. To provide a good experience, the app must run at a 30 fps. While developing the app, you always test it on a benchmark that contains 100 frames. After a few days of performance tuning the app consistently meets the performance requirement on this benchmark. Excited, you go to show your boss! She starts walking around the office with the app on, testing out how it works. After a few minutes of running the app, she comes back to your desk and says, "You're going to have to try harder, the app is not fast enough – it's not running at 30 frames per second". No changes have been made to the app. What happened?

Solution: The continuously running app caused the device to heat up and the processor decreased its clock frequency, reducing your app's performance. Although your app was efficient enough to meet the performance requirement when running for a short period of time (about 3 seconds), is was not efficient enough to maintain the desired level of performance at lower clock rates. ***Grading note: a number of students provided answers that were very different from the content of the course. Unexpected systems-related answers may be given consideration for credit.***

C. (3 pts) Recall a basic test and test-and-set lock, written below using compare and swap (`atomicCAS`) Keep in mind that `atomicCAS` is atomic although it is written in C-code below.

```
int atomicCAS(int* addr, int compare, int val) {
  int old = *addr;
  *addr = (old == compare) ? val : old;
  return old;
}

void LOCK(int* lock) {
   while (1) {
      if (*lock == 0)
         if (atomicCAS(lock, 0, 1) == 0)
            return;
   }
}
```

Imagine a program that uses this lock to synchronize access to a shared variable by 32 threads. This program is run on a processor with **four cores**, each of which are **eight-way multi-threaded** (32 total execution contexts across the machine.) You can assume that **the lock is highly contended, the cost of lock acquisition on lock release is insignificant and that the size of the critical section is large (say, 100,000's of cycles)**. You can also assume there are no other programs running on the machine.

Your friend (correctly) points there is a performance issue with your implementation and it might be advisable to not use a spin lock in this case, and instead use a lock that de-schedules a thread off an execution context instead of busy waiting. You look at him, puzzled, mentioning that the test-and-test-and-set lock means all the waiting threads are just spinning on a local copy of the lock in their cache, so they generate no memory traffic. What is the problem he is referring to? **(A full credit answer will mention a fix that specifically mentions what waiting threads to deschedule.)**

Solution: The problem here is that the thread with the lock must share the core with seven other threads that are also mapped to the core's execution slots (7/8 of the core's instruction processing ability is spent spinning!). It would be ideal to deschedule the three threads mapped to the same core as the thread holding the lock, so the thread in the critical section could have all the core's execution resources and complete the critical section as quickly as possible.

D. (3 pts) Given `atomicCAS` please implement a lock-free `atomicOR` below. (You do not need to worry about the ABA problem.) **In this problem, have atomicCAS return the result AFTER the OR occurs.**

```
// atomically bitwise OR the value val with the current value in addr
int atomicOR(int* addr, int val) {
   while (1) {
     int old = *addr;
     int new = old | val;
     if (atomicCAS(addr, old, new) == old)
        return new;
   }
}
```

E. (4 pts) Consider an application whose runtime on a single-core processor of type Core A consists of 80% deep neural network (DNN) evaluation and 20% graph processing. **All the DNN work must complete before the graph processing work is started (graph processing depends on the DNN output, so they cannot be done in parallel). However, each stage itself is perfectly parallelizable.** Imagine there are two possible cores, each with different performance characteristics on the DNN and graph processing workloads. The chart below gives the performance of Core B relative to core A on the two workloads (these are relative throughputs).

	Core Type	
	Core A	Core B
Resource Cost	1	1
Perf (throughput): DNN Eval	1	4
Perf (throughput): Graph Processing	1	0.5

If you had to design a multi-core processor for this workload that could have any mixture of cores so long as the total resource cost did not exceed 4, what mixture would you choose? **In your answer state (approximately) what speedup over a processor with one core of type A would you expect?** Hint: In each step of the problem its wise to parallelize work across all the chosen cores.

Math hint: consider a workload that spends 800 ms in DNN eval and 200 ms in graph processing on a single Core A. How much time would the computation take on a quad-core Core B processor?

Also, since the math in this problem isn't the cleanest... $200/2.5 = 80$, $200/3 \approx 67$, $800/13 \approx 62$

Solution: You want 1 core of type A and 3 cores of type B. The resulting cost is $\frac{800}{1+3\times4} + \frac{200}{1+3\times.5} = \frac{800}{13} + \frac{200}{2.5} = 62 + 80 = 142$. *Therefore the speedup is* $\frac{1000}{142} \approx 7$. *It's useful to note that 4 cores of type B gives a cost of* $\frac{800}{16} + \frac{200}{2} = 50 + 100 = 150$, *which is higher than the heterogeneous case.*

F. (3 pts) Consider a parallel version of the 2D grid solver problem from class. The implementation uses a 2D tiled assignment of grid cells to processors. (Recall the grid solver updates all the red cells of the grid based on the value of each cell's neighbors, then all the black cells). Since the grid solver requires communication between processors, you choose to buy a computer with a cross-bar interconnect. Your friend observes your purchase and suggests there there is another network topology that would have provided the same performance at a lower cost. What is it? (Why is the performance the same?)

Solution: In this application, each processor need only communicate with the processor that responsible for its left, right, top, bottom grid neighbors. The mesh topology provides all of these connections as point-to-point links, so the app will run just as well on a mesh as on a fully-connected crossbar. The mesh has $O(N)$ *cost, where the crossbar has* $O(N^2)$*, where N is the number of processors. A 2D torus would also be a valid answer.*

Warm Up: More Miscellaneous Short Problems

Problem 2. (22 points):

A. (3 pts) Google has created the Tensor Processing Unit (TPU), a specialized processor for accelerating machine learning computations, especially for evaluating deep neural networks (DNNs). Give one technical reason why DNN evaluation is a workload that is well suited for fixed-function acceleration. **Caution: be precise about what aspect(s) of the workload are important! Your reason should not equally apply to parallel processing in general.**

Solution: As we discussed in class, DNN evaluation exhibits high arithmetic intensity (a well-written convolution is compute bound on modern CPUs and GPUs), so it would benefit from hardware that provides high-throughput, low-energy arithmetic capability. Since DNN costs are dominated by a small number of operations (kernels like dense convolution), accelerating these specific operations in HW yield efficient performance. (Note: answers such as DNNs require "lots of computation", or access many parameters, correctly state properties of modern deep networks, but didn't identify a reason why an ASIC is a reasonable solution. Bandwidth heavy applications are poor candidates for ASIC acceleration, and parallelism alone doesn't justify a fixed-function solution.

B. (3 pts) Most of the domain-specific framework examples we discussed in class (Halide, Liszt, Spark, etc.) provide **declarative abstractions** for describing key performance-critical operations (processing pixels in an image, iterating over nodes in a graph, etc). Give one performance-related reason why the approach of tasking the application programmer with specifying "what to do", rather than "how to do" it, can be a good idea.

Solution: Declarative abstractions leave implementation details up to the system implementer, so the implementation can be specialized to a particular hardware architecture (whether or not to parallelize, when to use fixed-function hardware, etc.). For example, in class we discussed how a declarative approach to specifying operations on a graph could lend itself to very different implementation on clusters of CPUs vs. a GPU. Similar examples were given for image processing.

C. (3 pts) Consider the implementation of `unlock(int* x)` where the state of the lock is *unlocked* when the lock integer has the value 0, and locked otherwise. You are given two additional functions:

```
void write_fence();  // all writes by the thread prior to this operation are
                     // guaranteed to have committed upon return of this function

void read_fence();   // all reads by the thread prior to this operation are
                     // guaranteed to have committed upon return of this function
```

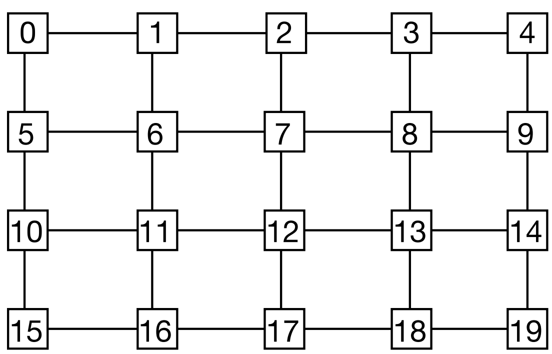
Assume the memory system is a **provides only relaxed memory consistency** where read-after-write (W→R) ordering of memory operations is not guaranteed. (It is not true that writes by thread T must commit before later (independent) reads by that thread.) Provide a correct implementation of `unlock(int* x)` that uses the minimal number of memory fences. **Please also justify why your solution is correct... using the word "commit" in your answer might be a useful idea.**

Solution:

```
void unlock(int* x) {*x = 0;}
```

We require that all writes in the critical section commit (recall that commit means be observable by other processors) prior to the commit of the write to the lock variable to perform the unlock. This ordering is already preserved by the system so no fences are necessary.

Note that in this relaxed memory system, an implementation of `lock(int* x)` *would need to ensure that reads after the lock (in the critical section) were not moved up in front of the write that took the lock. Therefore we need to use a read fence in the lock, but this was not part of the question.*



D. (3 pts) You may recall from class that the Xeon Phi processor uses a mesh network with "YX" message routing. (Messages move vertically in the mesh, undergo at most one "turn", and then move horizontally through the network. Consider two messages being sent on the 20 node mesh shown above. Both messages are sent at the same time. Each link in the network is capable of transmitting one byte of data per clock. Message 1 is sent from node 0 to node 14. Message 2 is sent from node 11 to node 13. **Both messages contain two packets of information and each packet is 4 bytes in size.**

Assume that the system uses store-and-forward routing. You friend looks at message workload and says "Oh shoot, it looks like we're going to need a more complicated routing scheme to avoid contention." Do you agree or disagree? Why?

Solution: Contention does not exist. It will take $3\times4=12$ cycles before the first packet from Message 1 arrives at node 11. However, after 8 cycles the Message 2 has already completely been transmitted over this link, so the link is free.

E. (4 pts) Now consider the same setup at the previous problem, except the network is modified to use wormhole flow-control with a flit size of 1 byte. (1 flit can be transmitted per link per cycle.) Is there now contention in the network? **Assuming that Message 1 has priority over message 2 in the network, what is the final latency of the end-to-end transmission of Message 2?**

Solution: Three flits of Message 2 leave Node 11 before the first flit of Message 1 arrives. In clock cycle 4 the first flit of message 1 starts transmitting over the 11-12 link. (They have priority over flits from message 2). The link becomes available again 8 cycles later in cycle 12. Cycles 12-16, the final 5 flits of Message 2 leave node 11. With the last flit getting to node 13 in cycle 17. So the total latency is 17 cycles. (A simple solution computes the answer 17 from 8+9. Under no contention, message 2 would have taken 9 cycles to transmit with cut-through flow control. It gets delayed for 8 cycles by message 1.)

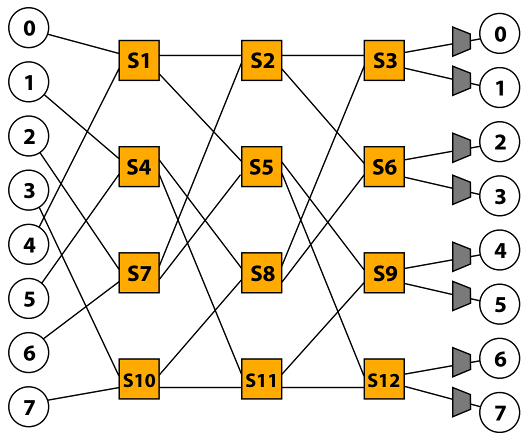
F. (3 pts) You are asked to implement a version of transactional memory that is both **eager and pessimistic**. Given this is a pessimistic system, on each load/store in a transaction T0, the system must check for conflicts with other pending transactions on other cores (let's call them T1, T2). Give a very brief sketch of how the system might go about performing a conflict check for a READ by T0. (A sentence or two about what data structure to check is fine.)

Solution: For each read by T0, the system needs to check for writes to the same address in the undo log of transactions T1 and T2. Any solution that mentioned checking the undo log of other transactions as given full credit.

Interconnects

Problem 3. (9 points):

Consider sending two 256-bit packets in the Omega network below. Packet A is sent from node 4 to node 0, and packet B from node 6 to node 1. The network uses worm-hole routing with flits of size 32 bits. All network links can transmit 32 bits in a clock. **The first flit of both packets leaves the respective sending node at the same time. If flits from A and B must contend for a link, flits from packet A always get priority.**



A. (2 pts) What is the latency of transmitting packet A to its destination?

Solution: 11 cycles. 4 cycles for first flit to arrive. 7 additional cycles for the rest of the packet.

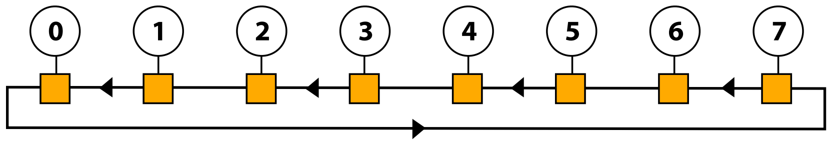
B. (3 pts) What is the latency of transmitting packet B to its destination? Please describe your calculation– switches are numbered to help your explanation.)

Solution: 19 cycles. The first flit gets to its destination 4 + 8 cycles (it must wait for 8 cycles for the shared link at switch S2). Then the rest of the packet arrives over the next 7 cycles.

C. (2 pts) If the network used store-and-forward routing, what would be the minimum latency transmitting one packet through the network? (Assume this packet is the only traffic on the network.)

Solution: 32 cycles. 8 cycles to transmit the packet over a link × four links between any two nodes on the network.

D. (2 pts) Now consider sending packet A from node 2 to node 0 and packet B from node 5 to 3 on the unidirectional ring interconnect shown below. Assuming the conditions from part A (32-bits send over a link per clock, worm-hole routing, same packet and flit sizes, both messages sent at the same time, packet A prioritized over packet B), what is the minimum latency for message A to arrive at its destination? Message B?



Solution: 9 cycles (2 cycles for the first flit to arrive, 7 more for the remainder of the packet.

Making Hash Tables Great Again

Problem 4. (10 points):

Note: This problem is from Spring, 2016, before the most recent presidential election. What was intended to be humorous then might not seem so funny now.

Hard times have fallen on the hash table industry. The dominate company TrustyHash used to have the world's most popular hash table implementation (given below), but the implementation is not thread-safe and cannot be used with modern multi-core processors. Specifically, the hash table has an `tableInsert` function that takes two strings, and inserts both strings into the table **only if neither string already exists in the table**.

```
struct Node {
  string value;
  Node* next;
};

struct HashTable {
   Node* bins[NUM_BINS];  // each bin is a singly-linked list
};

int   hashFunction(string str);           // maps strings uniformly to [0-NUM_BINS]
bool  findInList(Node* n, string str);    // return true is str is in the list
void  insertToList(Node* n, string str);  // insert str into the list

bool tableInsert(HashTable* table, string s1, string s2) {
   int idx1 = hashFunction(s1);
   int idx2 = hashFunction(s2);

   if (!findInList(table->bins[idx1], s1) &&
       !findInList(table->bins[idx2], s2)) {
     insertToList(table->bins[idx1], s1);
     insertToList(table->bins[idx2], s2);
     return true;
   }

   return false;
}
```

The powerful hash table SuperPAC *Citizens for O(1) Lookup* looks to this year's presidential candidates for help. Always confident, Donald Trump steps into the fray and announces "Let me tell you what I'm going to do. I'm going to put a great big lock around the entire table. Two threads WILL NOT end up operating on my version of the hash table at the same time that's for sure. A thread will have to get that lock before coming in. We are going to make this hash table (THREAD) SAFE!"

Bernie Sanders, trying to win support for his campaign, says "Donald, you've got it all wrong. Hashing distributes elements evenly throughout the key space. Distribution is good (for parallelism). I'm a big fan of hashing!"

(Question on next page...)

Citizens for O(1) Lookup is intrigued, but is concerned that Bernie's plan is a little low on implementation details. Please implement `tableInsert` below in a manner that enables maximum concurrency. (It has been copied on this page with space for clarity). You may add locks wherever you wish. (Please update the structs as needed.) Note, to keep things simple, your implementation SHOULD NOT attempt to achieve concurrency within an individual list (notice we didn't give you implementations for `findInList` and `insertInList`). **Note that like many election-year promises, things are a little more complex than they seem. You should assume nothing about `hashFunction` other than it distributes strings uniformly across the 0 to `NUM_BINS` domain. Keep in mind common pitfalls of fine-grained locking.**

```
struct Node {
  string value;
  Node* next;
};

struct HashTable {
   Node* bins[NUM_BINS];      // each bin is a singly-linked list
   Lock  binLocks[NUM_BINS];  // lock per bin
};

int   hashFunction(string str);           // maps strings uniformly to [0-NUM_BINS]
bool  findInList(Node* n, string str);    // return true is str is in the list
void  insertInList(Node* n, string str);  // insert str into the list

bool tableInsert(HashTable* table, string s1, string s2) {
   int idx1 = hashFunction(s1);
   int idx2 = hashFunction(s2);
   bool onlyOne = false;

   // be careful to avoid deadlock due to (1) creating a circular wait or
   // (2) due to the same thread taking the same lock twice
   if (idx1 < idx2) {
     lock(binLocks[idx1];
     lock(binLocks[idx2];
   } else if (idx1 > idx2) {
     lock(binLocks[idx2];
     lock(binLocks[idx1];
   } else {
     lock(binlocks[idx1];
     onlyOne = true;
   }

   if (!findInList(table->bins[idx1], s1) &&
       !findInList(table->bins[idx2], s2)) {
     insertToList(table->bins[idx1], s1);
     insertToList(table->bins[idx2], s2);

     unlock(binLocks[idx1]);
     if (!onlyOne)
       unlock(binLocks[idx2];
     return true;
   }

   unlock(binLocks[idx1]);
   if (!onlyOne)
     unlock(binLocks[idx2];
   return false;
}
```

Comparing and Swapping

Problem 5. (18 points):

The logic of atomic compare-and-swap is given below (Keep in mind that atomic CAS is an operation that carries this sequence of logic **atomically**.)

```
int CAS(int* addr, int compare, int val) {
   int old = *addr;
   *addr = (old == compare) ? val : old;
   return old;
}
```

Consider a program where multiple threads cooperate to compute the sum of a large list of **SIGNED INTEGERS**.

```
// global variables shared by all threads

int values[N];  // assume is very large
int sum = 0;

///////////////////////////////////////////////////////

// per thread logic (assume thread_id, num_threads are defined as expected)

for (int i=thread_id; i<N; i+=num_threads) {
  sum += values[i];
}
```

A. (5 pts) There is a correctness problem with the above C code when `num_threads > 1`. Using CAS, please provide a fix so that the code computes a correct parallel sum. To make this problem a little trickier, there are two rules: **(1) You must provide a non-blocking (lock-free) solution. (2) Each iteration of the loop should update the global variable `sum`. You are not allowed to accumulate a partial sum locally and reduce the partial sums after the loop.**

   Solution:

```
for (int i=thread_id; i<N; i+=num_threads) {
  while (1) {
    int old = sum;
    int new = old + values[i];
    if (CAS(&sum, old, new) == old)
       break;
  }
}
```

B. (5 pts) In the original code above, `sum += values[i]` is a read-modify-write operation. (The code reads the value of `sum`, computes a new value, then updates the value of `sum`. One way to fix the code above is to make this sequence of operations atomic (i.e., no other writes to `sum` occur in between the read and the write by one thread. **Does your solution in Part A maintain atomicity of the read-modify-write of `sum`?** (why or why not?) If it does not, why are you confident your solution is correct? **Keep in mind the numbers to be summed are signed integers.**

   Solution: In this problem, CAS can succeed even if other operations have come in between the read and the write to `sum`, provided those operations in total, resulted in no net change to `sum` (e.g., imagine the operations: +x, and -x). Therefore, solutions have used CAS to correctly avoid race conditions that could impact the sum.

C. (6 pts) Now imagine the problem is changed slightly to include a new shared variable `count`, representing the number of inputs that have been summed together to yield the value in `sum`. `count` and `sum` must always stay in sync. **In other words, it should not be possible to ever observe a value of `count` and `sum` such that the sum of the first `count` elements of `vals` don't add up to `sum`.**

```
// global variables shared by all threads
int values[N];
int sum = 0;
int count = 0;

////////////////////////////////////////////////////////

// per thread logic
for (int i=thread_id; i<N; i+=num_threads) {
  sum += values[i];
  count++;
}
```

Using only CAS, please provide a correct, thread safe version of the code. **This problem is independent of parts A and B.** However like part A, you must update the shared variables each iteration of the loop (no local partial sums). **Unlike part A, you can take any approach to synchronization, even if it IS NOT LOCK FREE.**

Solution: The most straightforward solution was to just build a lock using CAS, then wrap the increment of count and sum with the lock. Another solution a few students provided was to wait until the count matched the variable `i` *in the loop, which eliminated any need for any locks or a CAS. This solution can be thought of as a ticket lock where the tickets have already been pre-assigned, so the only thing left to do is release the "lock".*

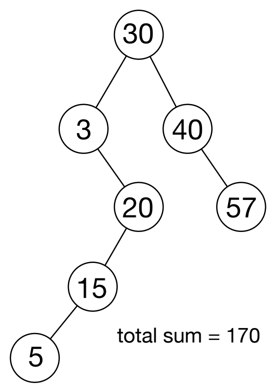
D. (2 pts) Imagine you had a double-word CAS (operating on 64-bit values in memory), implement a lock-free solution to part C. **Also answer the following question: is the shared variable update now guaranteed to be atomic? (why or why not?)**. For simplicity, assume double-word CAS returns true on success, false otherwise.

Solution: Doubleword CAS makes a solution essentially as easy as part A. Note that since count increases monotonically, a CAS will not ever succeed if other operations have been interspersed. So CAS success very guarantees the update is atomic.

Transactions on Trees

Problem 6. (18 points):

Consider the binary search tree illustrated below.



The operations `insert` (insert value into tree, assuming no duplicates) and `sum` (return the sum of all elements in the tree) are implemented as transactional operations on the tree as shown below.

```
struct Node {
   Node *left, *right;
   int value;
};
Node* root;  // root of tree, assume non-null

void insertNode(Node* n, int value) {
  if (value < n) {
    if (n->left == NULL)
       n->left = createNode(value);
    else
       insertNode(n->left, value);
  } else if (value > n) {
    if (n->right == NULL)
       n->right = createNode(value);
    else
       insertNode(n->right, value);
  }  // insert won't be called with a duplicate element, so there's no else case
}

int sumNode(Node* n) {
   if (n == null) return 0;
   int total = n->value;
   total += sumNode(n->left);
   total += sumNode(n->right);
   return total;
}

void insert(int value) {
  atomic { insertNode(root, value); }
}

int sum() {
   atomic { return sumNode(root); )
}
```

Consider the following four operations are executed against the tree in parallel by different threads.

```
insert(10);
insert(25);
insert(24);
int x = sum();
```

A. (3 pts) Consider different orderings of how these four operations could be evaluated. Please draw all possible trees that may result from execution of these four transactions. (Note: it's fine to draw only subtrees rooted at node 20 since that's the only part of the tree that's effected.)

There are only two. 20 $\rightarrow$ 25 $\rightarrow$ 24 or 20 $\rightarrow$ 24 $\rightarrow$ 25

B. (2 pts) Please list all possible values that may be returned by `sum()`.

Solution: 170, 180, 194, 195, 204, 205, 219, 229

C. (2 pts) Do your answers to parts A or B change depending on whether the implementation of transactions is optimistic or pessimistic? Why or why not?

Solution: Definitely not! The choice of how to ***implement*** *a transaction cannot change the* ***semantics*** *of the transactional abstraction.*

D. (3 pts) Consider an implementation of **lazy, optimistic** transactional memory that manages transactions at the granularity of tree nodes (the read and writes sets are lists of nodes). Assume that the transaction `insert(10)` commits when `insert(24)` and `insert(25)` are currently at node 20, and `sum()` is at node 40. Which of the four transactions (if any) are aborted? **Please describe why.**

Solution: Only `sum` is aborted since the write set of the committing transaction (which is node 5) conflicts with the read set of `sum`. Note that there is no conflict with the other insertions since they read no data written by `insert(10)`.

E. (3 pts) Assume that the transaction `insert(25)` commits when `insert(10)` is at node 15, `insert(24)` has already modified the tree but not yet committed , and `sum()` is at node 3. Which transactions (if any) are aborted? **Again, please describe why.**

Solution: In this case, `insert(10)` is aborted since 20 is in its read set, and it was modified by the committing transaction. (This answer relies on the fact that transactions are managed at the level of tree nodes. If they were managed at the level of words, then there would be no need to abort, since the only value written by `insert(25)` is the right pointer of node 20, and this was not read by `insert(10)`.) `insert(24)` is also aborted since its read and write sets conflict with the write set of the committing transaction. `sum` is not aborted since it hasn't progressed enough to reach node 20.

F. (3 pts) Now consider a transactional implementation that is **pessimistic** with respect to writes (check for conflict on write) and **optimistic** with respect to reads. The implementation also employs a "writer wins" conflict management scheme – meaning that the transaction issuing a conflicting write will not be aborted (the other conflicting transaction will). Describe how a **livelock problem** could occur in this code.

Solution: The problem is that `insert(24)` can write to `n->right` of node 20, which conflicts with `insert(25)`'s read/write of the same node. `insert(25)` will abort and restart, and then it's own update of `n->right` on the retry will cause `insert(24)` to abort if that transaction did not have time to commit.

G. (2 pts) Give one livelock avoidance technique that an implementation of a pessimistic transactional memory system might use. You only need to summarize a basic approach, but make sure your answer is clear enough to refer to how you'd schedule the *transactions*.

Solution: Any standard answer from the implementation of locks lecture would work, but in this context the solutions are used for implementing transactions, not locks: you could try random backoff, give priority to a transaction that's been aborted too many times in the past, put transactions that have aborted in a list and process the list serially (like a ticket lock), etc.

Two Box Blurs are Better Than One

Problem 7. (12 points):

An interesting fact is that repeatedly convolving an image with a box filter (a filter kernel with equal weights, such as the one often used in class) is equivalent to convolving the image with a Gaussian filter. Consider the program below, which runs two iterations of box blur.

```
float input[HEIGHT][WIDTH];
float temp[HEIGHT][WIDTH];
float output[HEIGHT][WIDTH];

float weight;   // assume initialized to (1/FILTER_SIZE)^2

void convolve(float output[HEIGHT][WIDTH], float input[HEIGHT][WIDTH], float weight) {

  for (int j=0; j<HEIGHT; j++) {
    for (int i=0; i<WIDTH; i++) {
      float accum = 0.f;
      for (int jj=0; jj<FILTER_SIZE; jj++) {
        for (int ii=0; ii<FILTER_SIZE; ii++) {

           // ignore out-of-bounds accesses (assume indexing off the end of image is
           // handled by special case boundary code (not shown)

           // count as one math op (one multiply add)
           accum += weight * input[j-FILTER_SIZE/2+jj][i-FILTER_SIZE/2+ii];
        }
      }
      output[j][i] = accum;
    }
  }
}

convolve(temp, input, weight);
convolve(output, temp, weight);
```

A. (2 pts) Assume the code above is run on a processor that can comfortably store `FILTER_SIZE*WIDTH` elements of an image in cache, so that when executing `convolve` each element in the input array is loaded from memory exactly once. What is the arithmetic intensity of the program, in units of math operations per element load?

Solution: It is $\texttt{FILTER\_SIZE}^2/2$*, since each input and output pixel are read exactly once, and each* `convolve` *operation performs* $\texttt{FILTER\_SIZE}^2$ *operations per pixel. We also accepted* $\texttt{FILTER\_SIZE}^2$ *for full credit since the question referred to "per element load".*

It's been emphasized in class that it's important to increase arithmetic intensity by exploiting producer-consumer locality. But sometimes it is tricky to do so. Consider an implementation that attempts to double arithmetic intensity of the program above by producing 2D chunks of output at a time. Specifically the loop nest would be changed to the following, **which now evaluates BOTH CONVOLUTIONS.**

```
for (int j=0; j<HEIGHT; j+=CHUNK_SIZE) {
  for (int i=0; i<WIDTH; i+=CHUNK_SIZE) {

    float temp[..][..];  // you must compute the size of this allocation in 6B

    // compute required elements of temp here (via convolution on region of input)

    // Note how elements in the range temp[0][0] -- temp[FILTER_SIZE-1][FILTER_SIZE-1] are the tem
    // inputs needed to compute the top-left corner pixel of this chunk

    for (int chunkj=0; chunkj<CHUNK_SIZE; chunkj++) {
      for (int chunki=0; chunki<CHUNK_SIZE; chunki++) {
        int iidx = i + chunki;
        int jidx = j + chunkj;
        float accum = 0.f;
        for (int jj=0; jj<FILTER_SIZE; jj++) {
          for (int ii=0; ii<FILTER_SIZE; ii++) {
            accum += weight * temp[chunkj+jj][chunki+ii];
          }
        }
        output[jidx][iidx] = accum;
      }
    }
  }
}
```

B. (2 pts) Give an expression for the number of elements in the `temp` allocation.

Solution: `(CHUNK_SIZE+FILTER_SIZE-1)`^{2}. *We also accepted* `(CHUNK_SIZE+FILTER_SIZE)`^{2} *for full credit.*

C. (3 pts) Assuming `CHUNK_SIZE` is 8 and `FILTER_SIZE` is 5, give an expression of the **total amount of arithmetic performed per pixel of output** in the code above. You do not need to reduce the expression to a numeric value.

Solution: Need $12 \times 12 = 144$ *elements of* `temp` = $5 \times 5 \times 144 = 3600$ *operations. Producing 64 elements of* `output` *is another* $64 \times 25 = 1600$ *operations. So there are now* $\frac{3600+1600}{64} = \frac{5200}{64} \approx 81$ *operations per pixel, compared to* $2 \times 25 = 50$ *operations per pixel in part A.*

D. (3 pts) Will the transformation given above improve or hurt performance if the original program from part A was *compute bound* for this `FILTER_SIZE`? Why?

Solution: It will hurt performance since it increases the number of arithmetic operations that need to be performed, and the program is already compute bound. Note that a fair number of students said that the problem is was ***arithmetic intensity was increased****, hence the slowdown. Increasing arithmetic intensity of a compute-bound program will not change its runtime if the total amount of work stays the same (it just reduces memory traffic). The essence of the answer here is that more work is being done.*

E. (2 pts) Why might the chunking transformation described above be a useful transformation in a mobile processing setting regardless of whether or not it impacts performance?

Solution: Since the energy cost of data transfer to/from DRAM is significantly higher than the cost of performing an arithmetic operation, reducing the amount of data transfer is likely to reduce the energy cost of running the program. Some students mentioned that reduced memory footprint was also a nice property of the transformed program (it doesn't have to allocate `temp`*), particular since DRAM sizes are smaller on mobile devices. We also accepted this answer for credit.*