Name: ____________________________________ unique name: ______________

Sign the honor code:

I have neither given nor received aid on this exam nor observed anyone else doing so.

______________________________

Scores:

<table>
<thead>
<tr>
<th></th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/15</td>
</tr>
<tr>
<td>2</td>
<td>/10</td>
</tr>
<tr>
<td>3</td>
<td>/20</td>
</tr>
<tr>
<td>4</td>
<td>/25</td>
</tr>
<tr>
<td>5</td>
<td>/30</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>/100</td>
</tr>
</tbody>
</table>

NOTES:

- Open lecture notes and open papers assigned for reading.
- Calculators are allowed.
- **Do not use web sources.**
- Be sure to show work and explain what you’ve done when asked to do so.
1) **Short Questions [15 pts]**

a. Can transactional memory be used to replace barrier synchronizations? Explain. [3 pts]

TM provides mutual exclusion, but can't be used to ensure a specific order between two threads. Barriers do provide point-to-point ordering between a group of threads. So you can't replace barriers with TM entirely. But you can optimize a legacy barrier's performance using TM -- speculate past a barrier, and recover on conflict.

b. Is it possible for a concurrent program that uses transactions to perform worse than its single-threaded execution? Explain with an example, if necessary. [3 pts]

Yes. TM conflicts entail rollback-and-re-execution, which results in wasted compute work. Also, conflict detection and version management incurs overhead. If these overheads are not mitigated through performance gains from optimistic concurrency, then TM will perform worse. Example: transactions in all threads read-modify the same variable.

c. Describe briefly how GPUs enable fast warp-level context switches. Specify two main reasons why a similar functionality (fast thread context switch) is less beneficial for CPUs, so much so that none of the commercially available CPU hardware have that support. [6 pts]

GPUs have large registers that store the architectural register states of all "live" warps scheduled on a GPU core. This enables the hardware to context switch from one warp to another, without having to save (old thread's) and restore (new thread's) register states from memory through a software OS routine.

CPU programs would also benefit from fast context switches, provided it exhibits similar characteristics as GPU programs: lots of threads (highly parallelizable), low data reuse, throughput-oriented, and not latency sensitive. But these properties are not as common in CPU programs compared to GPU programs. CPU is therefore optimized for latency with large caches and out-of-order execution to hide memory latency, instead of switching fast between threads. (Hyperthreaded CPU (or SMT) deviates from this design philosophy by enabling simultaneously execution of several threads on a core to increase throughput and perform useful work even when a thread is stalled due to cache miss).

Some answered by stating the reasons for why context switches in CPUs are expensive. Or pointed out that they are rare, and so fast context switch is not useful. But the question is, what if you had a fast context switch like in GPU that allowed you to hide memory latency by switching between threads (which would increase CS frequency)? Why would that be less
beneficial? Some of you said that register files for CPUs would be larger than what you would need for GPUs. But the architectural state of x86 has only few (8) registers per thread.

d. What is the main difference between SIMD vs SIMT from a programming perspective? [3 pts]

SIMD - Single thread with vector instructions, where a vector instruction synchronously operate on pre-defined set of data elements.
SIMT - Multiple identical threads that may differ in their control flow. Compiler/hardware groups them into warps, and only the threads within a warp is executed synchronously.

2. Parallelism [10 points]

(a) Say we have an application with three steps that run sequentially - step A, step B, and step C each take 35%, 25%, and 40% of the total program runtime respectively. You read the source code and realize that each step can be parallelized. But due to how the data is organized there are limits to how many cores each step can effectively use:

Step A can run on at most 8 processors
Step B can run on at most 10 processors
Step C can run on at most 4 processors.

If you only have time to optimize one of these steps, which would you choose and why? [4 points]

A, use Ahmdal’s law to prove this.

(b) Say we had a programmable multi-core processor that could reconfigure itself in a negligible amount of time to change its number and size of cores. Assume two valid configurations:

Big-Few: Few big cores
Small-Many: Many small cores

A data-center needs to process tasks with different QoS constraints (specified in terms of latency and throughput). Determine a policy that a “controller” in a data-center can use to configure its programmable multi-core processors and schedule tasks on them. [6 points]

We assume information about the program based on different QoS constraints - high throughput tasks are given small many, and tight latency are given big few. Make sure we prioritize the tight
latency requirements and give big few in the case where we have both high throughput and tight latency.
3) Cache Coherency [20 points]

Upgrade Predictor

In shared-memory programs, accesses that are guarded by locks often exhibit migratory sharing patterns -- i.e., data is often read and modified on one processor before migrating to another processor that subsequently enters the critical section. Unfortunately, for migratory data, conventional coherence protocols are inefficient because the read to the data places the data into the "shared" state and the write requires an extra bus transaction to invalidate other copies, resulting in two consecutive bus transactions to read and write.

Design a table-based Upgrade Predictor that would identify such read/write sequences (i.e., an instruction PC pair) to the same address and upon future visits to the read, would upgrade the read to a write by sending a write request to the cache to eliminate the first bus read transaction.

(Hint 1: The tables should not be indexed by data addresses because index space would be huge. Hint 2: Only read/write sequences by one processor without intervening accesses by other processors are migratory.)

- Draw a diagram for your predictor (or describe in detail)
There were many different ways to answer this question correctly. Key ideas: use address tag for lookup, look for read miss->write miss pattern (coherency, NOT general read/write access to cache) in the learning process, when pattern is predicted instead of sending getS send getM. If we mispredict update the learning/upgrade process - explain why mispredictions may cause unnecessary invalidations but will not affect correctness.

- Describe the learning process
● Describe the lookup and upgrade prediction process

● What happens upon a misprediction?
4) Synchronization [25 points]

(a) Explain a scenario where spin based synchronization will be better than blocking based synchronization. [1 pt]
There are many correct answers: wait time is less than 2x context switch time, very low lock contention, etc.

(b) Explain a scenario where blocking-based synchronization is better than spin-based synchronization. [1 pt]
Again many correct answers: wait time is greater than 2x context switch time, high lock contention, etc.

(c) Improve the fairness of a standard Test-and-Test-and-Set lock implementation using $\text{Random}(x,y)$ which returns a random integer in the range of $[x,y]$. Assume $\text{Random}(x,y)$ is truly random and cheap. In addition, you can introduce only one additional global variable.

Rewrite the code below and explain your changes. [11 points]

// Test-and-Test-and-Set Skeleton Code
acquire(lock_ptr):
  while (true):
    // Perform “test”
    load [lock_ptr] -> original_value
    if (original_value == UNLOCKED):
      // Perform “test-and-set”
      old = compare_and_swap(lock_ptr, UNLOCKED, LOCKED)
      if (old == UNLOCKED):
        break // lock acquired!
    // keep spinning, back to top of while loop

release(lock_ptr):
  store[lock_ptr] <- UNLOCKED
// Your implementation

acquire(lock_ptr):
A somewhat popular, but not very elegant solution was to insert a wait/delay function with a random interval. How much fairness this actually grants is questionable, and many of you inserted this delay after the CAS meaning that if n threads tried to acquire the lock simultaneously this delay would not even occur during the first attempt to acquire the lock and there would be no increase in fairness! Regardless, it is fairly inefficient. Because the Random() function is supposed to be very cheap, a solution closer to this is more efficient and more fair:

global int N = 0;

acquire(lock_ptr):
    N++ // this can be done atomically or not, you have race conditions this way
    // but it should approximate the intended behavior well enough
    while (true):
        // Perform “test”
        load [lock_ptr] -> original_value
        // on average (N * 1/N) = 1 thread should gain access!
        if (original_value == UNLOCKED && Random(1, N) == 1):
            // Perform “test-and-set”
            old = compare_and_swap(lock_ptr, UNLOCKED, LOCKED)
            if (old == UNLOCKED):
                break // lock acquired!
            // keep spinning, back to top of while loop

release(lock_ptr):
    N--
    store[lock_ptr] <- UNLOCKED
(d) Analyze the following properties for your Randomized Test-and-Test-and-Set implementation, and compare it with test-and-test-and-set lock and ticket locks. \[12\text{ points}\]

These were graded relative to your answer from part c), many of you forgot to compare to both TTS and ticket locks.

i) **Performance**: Communication (coherence) overhead on a release during high lock contention:

ii) **Performance**: Acquire latency during low lock contention:

iii) **Fairness**: 
5. Transactional Memory for Detecting Data Races [30 pts]

(a) Say Intel releases a processor with hardware transactional memory (HTM-U) feature that supports unbounded transactions. Its design extends cache coherence protocol to detect conflicts. It handles unbounded transactions by serializing threads (executing one transaction at a time).

Unfortunately, all shared-memory multi-threaded software until now have been written using legacy synchronizations (locks, signal-wait, barriers, etc.). Obviously, they don’t use transactions, and so they will not benefit from HTM-U without additional effort.

Legacy parallel software may contain data-race concurrency bugs, and they will continue to be executed on legacy systems. Your goal is to develop an automatic debugging tool that uses HTM-U for finding data-races in legacy software as they manifest when these programs are executed. You have the freedom to change any part of the system - hardware, compiler, or runtime (OS). But make sure that any data-race that can manifest in the original system can manifest in the modified system -- this property is essential to make sure that your tool can find data-race bugs in the legacy program while running on legacy systems.

Definition of a dynamic data-race: A program execution has a data-race if two threads access the same memory location, at least one of those accesses is a write, and there is no happens-before order between those two accesses due to programmer defined synchronization in the original program (using legacy synchronization operations).

Sketch your main idea:
(Make sure to specify what constitutes a transaction in the modified legacy program)

Hardware changes, if any:
HTM's versioning and rollback-recovery is turned off. This helps discover more than one data-race within a conflicting SFR by continuing its synthetically constructed transaction that encapsulates it.

Compiler changes, if any:
(Hint: compiler allows you to emit extra code into the binary for analyzing a program when it executes):
Compiler places each *synchronization free region* (code region between any two synchronization operation in a given program) within a transaction. Before any synchronization operation it inserts a tx_end, and then after the sync operation, it inserts a tx_begin. Program (thread) start is instrumented with tx_begin, and program (thread) exit is instrumented with tx_end.

Any conflict detected between two concurrent transaction is a *potential* data race.
(Some of you placed entire thread within a transaction. This will report all concurrent accesses to a location as data-race, which is not useful. The goal is to find data-races in a legacy program written using locks, signal-wait and barriers.

Some of you placed only non critical sections within transactions. But data-race can exist between two critical sections protected by different locks).

OS changes, if any:

HTM invokes OS handler which then logs the conflicting instructions along with the cache block address that raised the conflict. (logging could done by hw or compiler)

Does your design guarantee to detect all data-races that manifest dynamically? Explain. False negatives: A data race may be missed when the conflicting transaction has either committed or yet to begin. Note that, there exists a data-race between two concurrent memory accesses as long as there is no happens-before order between them. Here is an example:

Thread-1          Thread-2

lock(L1)  lock(L3)
R1. x = 1  R3. x = 10
unlock(L1) unlock(L3)

lock(L2)  
R2. x= 2
unlock(L2)

R3 races with both R1 and R2. But at runtime, if "unlock(L3)" executes before "lock(L2)", no data-race between R3 and R2 would be detected, even though there is no happens-before order between them.

When unbounded transactions are serialized, HTM is effectively turned off, so we may also miss data-races within those transactions.
Does your design report any false data-races (data-races that did not manifest in the execution)? Explain.
False positives: Since the conflicts are detected by the HTM (by extending cache coherence protocol) at the cache block granularity, the proposed mechanism may report false data races due to false sharing.

(b) Say Intel’s HTM does not support unbounded transactions. How would this limit your capability in detecting data-races?

If HTM requires bounded transactions, then unbounded SFRs need to be broken into multiple bounded transactions. This may result in more data-races being missed. In the example above, if both critical sections were part of the same transaction, then the chances of finding both data-races is higher.

(c) If you want to help improve the correctness and performance of legacy software while executing on an HTM-U system, what automatic program transformations would you recommend? Be specific about how you propose to handle each of the legacy synchronization operations: locks, barriers, signal-wait.

Translate lock into tx_begin and unlock into tx_end. This will eliminate all data-races due to incorrect locking discipline (not ensuring that a variable's access is consistently protected by at least one lock).

SFRs outside critical sections (protected by lock/unlock) could be made transactions (similar to what we did for finding data-race bugs). This will reduce thread interleavings, thereby helping to mask any concurrency error that exists in the legacy program.

Barriers and signal-wait are left unmodified.