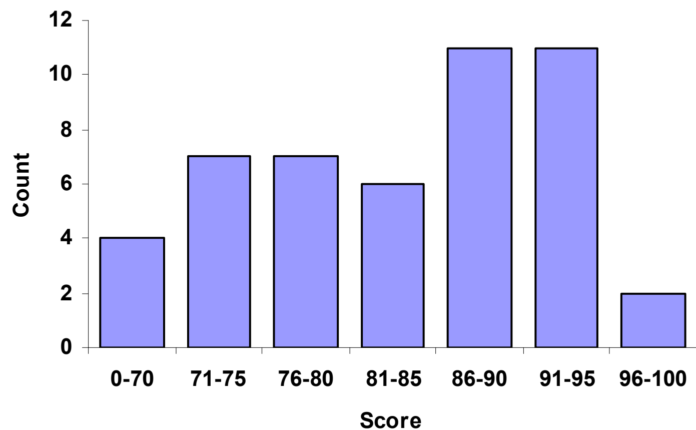
CS262a Midterm Answers
Fall 2008
Prof. Eric Brewer

Overall average: 83.8 out of 100
Standard deviation: 8.7.



(1) True/False
Average: 11.8 out of 14
Standard Deviation: 1.48

(1) True/False [14 points, 2 each] For each answer provide a one sentence explanation.

a) Aries could do "time travel" if you kept the whole log and a set of archived snapshots.

True. Time travel is the ability to go back to previous point in "serialization" time (which is well defined for ACID transactions). You can start from a fuzzy snapshot and roll forward to the target time using the redo log entries.

b) It is easy to add congestion control to UDP by implementing the sliding window protocol (from TCP) at user level.

False. It is possible to implement the protocol, but it is difficult to get the congestion control to work since the user-level process may not be scheduled for a long time.

c) LRVM would benefit from compiler support.

True. set-range calls are error prone and could be better enforced with compiler support (as mentioned in the paper).

d) Active messages are a good way to reach the maximum bandwidth of the underlying link.

False. Active messages are small and thus have a great deal of space overhead that limits their bandwidth (covered some in the U-Net paper).

e) A hypervisor could use lottery scheduling to allocate physical pages among guest OSs.

True. This is actually a decent idea, as the hypervisor needs some way to make allocation decisions, and lottery scheduling is a generic mechanism.

f) TCP can easily achieve the full bandwidth of a WiFi connection.

False. It is possible but not easy — even a little wireless noise will drop the bandwidth significantly, as TCP incorrectly believes there is congestion causing the losses.

g) If a thread dies while holding a lock, the OS should just release the lock.

False. The lock is protecting some invariant. Ideally, the OS would fix it first. Analogous to this is the Mesa policy of not releasing locks on thread abort if there is no exception handler ("unwind" code). Full credit if you mentioned "invariant" even if you argued to release the lock anyway.

(2) Flash Database

Average: 13.8 out of 16
Standard Deviation: 2.28

The goal is to create an ACID transactional database on top of Flash through the use of logging. Assume a block interface with 4K blocks on an "SSD" (solid-state drive). There is only one drive, so we need to use it for both the log and the main database store. Assume no seek or rotational delay.

a) [2] Given that we need not optimize for sequential writes, why do we need a log at all?

Still need it for non-trivial atomic updates, such as rename or some directory operations.

b) [4] One option would be to simply write out a page on commit ("force") and avoid having to have a redo log. Give two reasons why this is not a good tradeoff.

1) Need the redo log to do media recovery. Flash does lose data sometimes.
2) Multi-page transactions need to be atomic, which is not ensured with just atomic (single) page writes.

c) [4] One way to avoid undo log entries is to not allow pages to be written back until they are committed. Give two reasons why this is not a good tradeoff.

1) Memory pressure: some transactions may dirty many pages and thus reduce the usable memory for other transactions. In fact, the system may run out of memory and have to abort a transaction to get its pages back. In the worst case, such transaction can never be completed if it always needs more pages than will fit in memory.

2) With multiple overlapping transactions on the same page, that page may never be able to be written out, even though the individual transactions are committing. This makes the log very long and recovery very long.

d) [6] Assume that we do want pages to be written back (sometimes) before they commit. Explain how we can make this work without having undo log entries.

We just need to write the pages somewhere else (as done in LRVM). We could write them to a temporary file or even use the OS paging system to page out the buffer pages to the swap file. On commit, we need to move the page from its temporary location to the real location. We will need to remember that the page is in temporary storage even if we crash, so we can either label the backup file and scan it on recovery or we can write a log entry at the time of the steal that says where the page was stored. (This is not an undo log entry though, as it happens on steal once per page, rather than during updates.)

(3) Click Router

Average: 16.0 out of 20
Standard Deviation: 2.59

Figure 8 of the Click paper presents a standard IP router. Assume that the actual routing function, LookupIPRoute, is relatively expensive.
a) [3] Explain why the router, as is, will not perform that well on a 16-core many core chip.

As is, Click is single threaded!

b) [7] Assume that we can replicate LookupIPRoute without worrying about route updates. Using two new elements, Mux and DeMux, which are variants of queues with fan-in and fan-out respectively, explain how to change the Click graph so that the router will run well on 16 cores. The Click elements should be usable without change, although you may need more instances of some of them. If you need to change or create additional new elements, give the reasons.

There are many variations that did well. The one I had in mind was 1-2 cores for everything except LookupIPRoute (in the case of 2, they are "before" and "after"). A new demux operator should be something like a hash-function demultiplexor (as in Volcano) that hashes on the source or destination IP address, so that the router preserves the order of packets in a a stream. (This is not strictly necessary, but it will make TCP work much better; took off one point for not at least mentioning the reordering issue.) There needs to be a new queue somewhere to enable changing threads, but the best place is probably after the demux. Next, we need some kind of push-to-pull converter that has a thread (that is typically tied to a specific core) and that pull from the queue and pushes through LookupIPRoute and beyond. After the sequence of elements that has been replicated we need the new mux operator that takes k push inputs and builds one queue with a single pull output. Finally, the "after" thread pulls from the mux element and completes routing. At least the mux and demux elements have to be thread safe, everything else arguably has no shared state.

c) [10] Explain how to implement LookupIPRoute using a K42 clustered object so that it will have good performance on 16 cores and also handle route updates cleanly.

As updates are rare, the basic approach is to have local replicas on each core of the 14-15 cores that runs LookupIPRoute, so that lookups are entirely local. To have consistency, we need some kind of locking; one option would be readers/writers locking with the root getting a write lock when it needs to update the table. Another valid approach that is a little better is to make the local reps caches of the routing table; this requires less total space and works well if you have a flow-based hash function as suggested above, as that will mean you only need a subset of the table. In general, good partitioning tends to make better use of many independent caches, while simple load balancing tends to cause all caches to have roughly the same content (in some sense wasting cache space). For consistency in this case, it is sufficient to invalidate the cache entries during updates

Another, less good, answer is to "hot swap" the whole object on updates. This will atomically change the routing table so it is correct, but it is overly heavyweight.

(4) ReVirt

Average: 14.9 out of 18
Standard Deviation: 2.89

a) [3] Explain why ReVirt (as presented in class) does not work on multiprocessors.

True concurrency is a source of non-determinism that (normal) ReVirt does not handle.

b) [3] Assume that we have no pre-emption among threads (only yields/ blocking); does ReVirt then work correctly on multiprocessors?

This helps a little, but there may still be data races that are non-deterministically ordered.

c) [12, hard] Assume that we a SEDA server with one thread per stage and no shared variables except for the inter-stage queues. Explain how we can get replay to work using ReVirt even on a multiprocessor.

The key idea here is that we can use normal uniprocessor ReVirt to make a single stage deterministic; in this case, a SEDA stage is like a uniprocessor: it has one thread and no true concurrency (internally).

Thus the primary issue is how to make the queues replayable. Note that they don't strictly need to be deterministic, as long as whatever order occurs we can reliably replay. Note also that during replay, different cores may replay at different speeds even though they replay exactly the same instructions, so we can't assume that they will just play back in sync without some intervention.

The best answer is to modify the queues so that they log all the enqueue operations via ReVirt. Note that dequeue is only called by one thread, so once the order is set correctly by the enqueue operations, the dequeue order is also set. On replay, ReVirt must block the dequeue until the correct enqueue operation has arrived and then deliver that event. Thus given a replayable stage with the same exact input sequence, the stage is deterministic.

Strictly speaking the OS may use shared memory even if the SEDA code does not. It was fine to ignore this, since I specifically said that there are no shared variables. One way to handle it would be to run one VM on each core with its own OS. This ensures that there is no shared state among cores (and in fact is how ReVirt works naturally). The queues now need to move data between VMs, but since we already need to involve ReVirt on every enqueue, this is not a big change.

(5) Page Coloring and Memory Bandwidth

Average: 13.3 out of 16
Standard Deviation: 2.37

Page coloring is a technique that takes advantage of the simple mapping of physical pages to L2 cache lines; certain page ranges always map to a certain area of the L2 cache. Assume that there are 16 "colors", i.e. we can partition our pages into 16 groups each of which have 1/16 of the L2 cache. Cache lines can only evict other lines of the same color.

a) [2] Explain why this might make application performance more predictable.

We can isolate the cache for our application from other applications, thus eliminating a source of variance (i.e. less cache interference from other applications).

b) [4] Assume that we have so many cores that the primary bottleneck is off-chip memory bandwidth. Explain the general connection between page coloring and memory bandwidth.

For reads, memory bandwidth is entirely due to cache misses. So if an application has isolated cache space, then all of its misses and all of its read bandwidth are due to its actual demand.

c) [4] Explain how to use page coloring to allocate L2 cache space based on lottery tickets, assuming we have at most 16 ticket holders.

This question was less clean than I hoped, so it was graded leniently. With at most 16 ticket holders, we can enforce cache isolation by never sharing colors among apps (the OS likely needs a dedicated color, but it can be viewed as one of the 16 ticket holders). There a several acceptable approaches here, but the easiest is probably just to allocate colors with a normal lottery. This means that some ticket holders may not get any colors, and thus no cache space, and are thus not runnable. This is not philosophically different from standard lottery scheduling in which it may take many lotteries before a process gets to run. Apps with more tickets will get more colors and thus more cache space. Another variation involves using an inverse lottery to pick a color to take away from a running app to then re-lottery to another app. In all cases, it is worth pointing out that changing the color of a page requires actually copying it to a physical page of the right color (i.e. it is not just a remap operation). As an aside, the OS mechanism for colors is to have 16 free lists for pages, one for each color. When an app gets a new physical page, the OS pulls it off the corresponding free list.

With more that 16 ticket holders, you could either limit the system to at most 16 runnable at time (not good for 100 cores!) or you can reuse colors. In the latter case, the key point is that the more tickets you have the fewer apps should be sharing your colors.

d) [6] Explain how to combine lottery scheduling and page coloring to ensure fair allocation of memory bandwidth over time. There may assume counters that tell you how many cache misses were caused by the current thread.

This is very interesting question, to which I still don't really know the best answer. Grading focused more on the reasoning than on the specific answer (which is how real research works!). The core unknown is how to react if an app has a lot of cache misses: do you penalize it for using too much memory bandwidth by scheduling it less or reducing its cache space, or do you give it more cache space hoping to see an increase in hit rate and thus lower read memory bandwidth?

Also in general, if we are limited by memory bandwidth, we probably need to keep multiple apps in cache all the time, both because we have multiple cores, and because we can't quickly fill all of the caches for one app. Essentially, coloring makes it easier to partition cache space among cores or groups of cores.

The most important point is to have some feedback loop based on the cache miss rate so that you can alter scheduling and/or color allocation to bring it closer to its target value. In fact, just using a feedback loop and scheduling (and no color allocation) you can manage memory bandwidth: essentially you use compensation tickets schedule apps earlier if they didn't use much bandwidth and you actually temporarily reduce the tickets if they used too much. If a job used its whole quantum, but not much memory bandwidth, you might run it again immediately (since its cache is loaded). Similarly, on a multiprocessor, you probably want to co-schedule jobs of different colors (unless two jobs share memory).

With colors, we have the option or adding or removing a color from an app over time based on the feedback loop. To do this well, we'd really like to figure out how cache space relates to miss rate for each app. If we add cache space and see no decrease in miss rate (likely due to sequential patterns or a much bigger working set), we probably should go back to fewer colors and adjust scheduling instead. If the miss rate goes down, we should continue to consider adding colors over time. For apps with a low miss rate, we could either leave them alone, or even take away a color (unless we just added it); we should reward them not with more cache space, but with more execution time, ideally via a longer quantum.

(6) Extending MapReduce

Average: 14.1 out of 16
Standard Deviation: 1.94

As presented, MapReduce has a single map stage followed by a single reduction.

a) [3] The API actually supports multi-stage reduction. Explain how the API supports this and why it is useful.

The combiner function mentioned in the paper is a multi-stage reduction. It works because the output type is the same as the input type (for reduce). Chaining MapReduce operations is actually not quite the same as a multi-stage reduction.

b) [3] Extension 1 (shared data): We would like every M and R worker to have access to a read-only hash table, H, that maps from k3 -> v3, where k3 is the type of the key and v3 is the type of the value. H also provides an iterator. Explain how to implement this extension.

It depends on the size of the hash table. For small to medium size tables, we can just give each node its own local copy (since it is read only). This can be done most easily by having the MapReduce infrastructure preload the table from a file during initialization. For a really big hash table, we probably we could either partition it or put it on some servers that handle RPC requests for lookups (and cache results locally). If you did the RPC approach you need to explain how to handle iteration; one good way is to move large blocks of the table via RPC in one call, iterate through the block locally, and then get another block. A chord or DHT approach is not a good fit for this problem, as it is very expensive, we don't need updates, and the fault tolerance model doesn't really fit the simple restart model of MapReduce.

c) [6] Extension 2 (multi-stage MapReduce): We would like the reduce function to be able to initiate another explicit reduction phase, by performing a map-like generation of (k2, v2) pairs. Reduction phases continue until all

reducers decide not to initiate a new phase. Explain roughly how to implement this.

In general, we need to be able to move data around between iterations, so the basic approach is to create intermediate files (like the map function), pass their location to the master, and have the next iteration pull the data from these files. In the common case, the file will be local, but not always. Each iteration is independent and the master needs to make sure the whole iteration completes before moving on to the next one (again just like it does with the map stage). You also need to change the output type of reduce for the intermediate values so that it includes the keys for the next round. Since the master has to hear from all reducers, that communication can also include whether or not an additional round is needed. As described, the handling of the exit situation is unclear, so any answer was fine. It would probably be better to have an explicit flag for the last iteration, so that all nodes can perform any cleanup or conversion and output the final results.

d) [4] Using these two extensions, show how to use MapReduce to compute all-pairs shortest path for a large graph. You are given a set of weighted unidirectional edges to describe the graph; i.e. (source, dest, weight). It is sufficient to output the length of the shortest path for all pairs of (source, dest).

Multi-stage reduce should in general be able to do any recursive graph problem in parallel. The answer I had in mind was to divide the nodes among the reducers and have them do single-source shortest path for their nodes. Using Dijkstra's algorithm this means that for the kth iteration we find the all-pairs shortest paths (in parallel) that use at most k edges. This takes at most n iterations (for n nodes).

The read-only hash table can be used to hold the edges of the graph and their weights. The map function uses this to do the initial division into shortest paths of length 1.

In each reduce iteration, we try to extend all of the paths we have so far with one more hop and see if the new distance is shorter. We output our new set of shortest paths for our subset of the nodes. If we found no new shorter paths, then we are done, as future iterations will not find any either. After the last iteration, the output is the all-pairs shortest path.

Other algorithms, including Bellman-Ford (which handles negative weights), were fine as long as they actually did work in parallel and got the correct answer.