Homework 1 Solutions:

If we expand the square in the statistic, we get three terms that have to be summed for each i: (ExpectedFrequency[i]), (2ObservedFrequency[i]) and (ObservedFrequency[i])2 / Expected Frequency[i]. The first term sum can be precomputed whenever the expected frequencies are updated. The second term sums to 2, because the sums of the observed frequencies across all characters must sum to 1, if the observed frequencies are computed as ratios of observed occurences to length of field. The third term sums can be computed incrementally by keeping a counter S for the sum of the squares of the occurence of each character i. Computing S incrementally requires recording the current occurence count for each character i in array C[i] (as in the chapter).

When the chip sees character i, it looks up C[i], and increments S by 2C[i]+1. This is because going from x to (x+1)2 = x2+2x+1 causes an increment of 2x+1). The chip then increments C[i] by 1. When the field length L is determined because the end of the field is reached, S must be divided by L2 to normalized the counter from being a sum of occurrences (squared) to being the sum of frequencies (also squared). An even more efficient approach (replacing the division by a multiplication) is to pre compute the difference between the threshold and first and second terms, call it P. At the end, the chip should alarm if S > P · L2. Note that we got away with a single counter in this example because the Sum aggregate operator can be computed incrementally, just as the Max aggregate operator was computed incrementally in the chapter example.

Chapter 10

Question: ARP Caches: Another example of an exact match lookup is furnished by ARP caches in a router or end node. In an Internet router, when a packet first arrives to a destination, the router must store the packet and send an ARP request to the Ethernet containing the packet. The ARP request is broadcast to all end nodes on the Ethernet and contains the IP address of the destination. When the destination replies with an ARP reply containing the Ethernet address of the destination, the router stores the mapping in an ARP table, and sends the stored data packet with the destination Ethernet address filled in.

• What lookup algorithms can be used for ARP caches?
Solution: Any exact or even prefix matching algorithm can be used with ARP caches. In software, a simple hash function suffices with some chance of collisions and hence non-deterministic performance. In hardware, the ARP lookup in a router is often bundled in with prefix lookups, with the 32-bit ARP translation being a more specific match than the prefix for the subnet. The only disadvantage of this method is that it can increase the size of the prefix lookup table to include all stations on all LANs the router is connected to.

• Why might the task of storing data packets awaiting data translation result in packet reordering?
Solution: Typically, the task of actually sending an ARP request will not be done in the main forwarding fast path. Instead, the packet will be sent to a route processor which then issues the ARP and awaits a response. Any more packets coming in to the same address are also forwarded to the route processor where they are queued behind the first packet. When the response comes, the route processor has to add the ARP entry to the forwarding engine (s) in the fast path, and then forward the queue packets. The problem is to correctly order packets that arrive before the

ARP entry is made, and the packets that arrive after the ARP entry is made. If the route processor makes the ARP entry before sending the queued packets, packets arriving just after the ARP entry will find the ARP entry works, and will be forwarded on the fast path, bypassing the queued packets.

On the other hand, if the route processor first sends the queued packets before sending the ARP entry, the route processor may never finish its backlog of queued packets, as packets keep being queued as long as the ARP entry is not made. Worse, even if this backlog empties out, during the time the route processor is setting the ARP entry (which takes time), there may be forwarded packets that do not find the ARP entry set, and are in the process of being sent to the route processor. Since the route processor may be in a different line card, there can be considerable latency to communicate between route processor and forwarding engine which can hold further queued packets in transit. Once again, if there are packets in transit from the forwarding engine to the route processor, these packets can be reordered after packets that arrive just after the ARP entry is set. Avoiding this problem requires either that the forwarding engine halt all its work while it waits for an ARP entry to resolve (very wasteful) or complex interlocks between the route processor and forwarding engine.

• Some router implementations get around the reordering problem by dropping all data packets that arrive to find that the destination address is not in the ARP table (however, the ARP request is sent out). Explain the pros and cons of such a scheme.

Solution: Clearly, the advantage of such a scheme is that it avoids the reordering problem described above because there never is a backlog of queued packets: they are all dropped. Dropping certainly avoids the charge of reordering but it is a pretty Draconian solution!

However, the argument is that this often happens when a TCP SYN comes in to a station D that has been talked to in a long time. When that happens, the router drops the SYN, and the SYN is retransmitted. During the time of the retransmission, the ARP request generally gets resolved. Nevertheless, this still appears to be a violation of the ARP specification to make a less complex implementation possible. It costs at least one packet dropped every once in a while, and possibly leads to more dropped packets for UDP traffic. It appears that many routers do this "optimization".

Chapter 11 – Prefix Lookups

Question 1. Caching Prefixes: Suppose we have the prefixes 10*, 100*, and 1001*. Hugh Hopeful would like to cache prefixes instead of entire 32 bit addresses. Hugh's scheme keeps a set of prefixes in the cache (fast memory), in addition to the complete set of prefixes in slow memory. Hugh's scheme first does a best matching prefix search in the cache; if a matching prefix is found, the next hop of the prefix is used. If no matching prefix is found, a best matching prefix search is done for the entire database and the resulting prefix is cached. Periodically, prefixes that have not been matched for a while are flushed from the cache. Alyssa P. Hacker quickly gives Hugh a counterexample to show him that his scheme is flawed, and that caching prefixes is tricky (if not impossible). Can you?

Solution: The problem is that if a cached entry matches, there is no guarantee that there is no longer matching prefix in the forwarding table. Consider a table with two prefixes P1 = 00*, and P2 = 000*. Suppose a packet comes in whose destination address starts with 001. Then it matches P1, and the prefix P1 = 00* is cached. Next, a second packet comes in whose destination address starts with 000. When checked in the cache, it matches P1, and the packet will be forwarded wrongly using P1 when it should have been forwarded using P2. Caching can help as follows. When a cache entry hits, one can start directly with that entry in the forwarding table (especially useful in trie based schemes). In particular, an even easier solution is to cache prefixes if and only if there is no more specific prefix that extends this prefix. Since such extensions are not as common, this may work well. In general, however, one still has to do a prefix match in the cache, though this can be done using a small hardware CAM; worse, it is not clear that there are any worst-case improvements, which in the light of wire-speed requirements, makes prefix caching not attractive.

Question 2.Encoding prefixes in a constant length: We said in the text that encoding prefixes like 10*, 100*, and 1000* in a fixed length could not be done by padding prefixes with zeroes. It clearly can be done by padding with zeroes and adding an encoding of the prefix length. We want to study a more efficient method.

• How many possible prefixes on 32 bits can there be?
Solution: The number of prefixes of length 32 is 232, the number of length 31 is 231, and so on. Thus the total number of prefixes of any length is 232 + 231 + . . . + 1, which is 233 − 1.

• Show how to encode all such prefixes using a fixed length of 33 bits. Make sure that 10*, 100*, and 1000* encode to different values.
Solution: A simple solution is to code the prefix so that the boundary between the prefix and the don't care bits is signaled by the first 1 starting from the right. For example, imagine that we had to encode 10*, 100*, and 1000* in 5 bits (similar but longer encodings work for 32 bits). Then 1000* is encoded as 10001, 100* as 10010, and 10* as 10100. To decode, start from the right and look leftward for the first 1; remove all bits from (and including) this rightmost 1 and all bits to its right and replace all these bits by a *. For example, when applied to 10100, thus gives us 10* as desired.

• Can you use this fixed length encoding of prefixes to have the multiple hash tables used in Section ?? be packed into a single hash table? Why might this help in decreasing the chances of hash collisions for a given memory size?
Solution: It is a general principle of hash tables that aggregating multiple hash tables into one common memory gives the minimum collisions for a fixed amount of memory given for the whole solution. Using the fixed length encoding encodes all prefixes by a fixed 33-bit address. When searching for a prefix of length i, the lookup has to place a 1 in bit i and zeroes after that (up to bit 33) and then do a lookup in the common table. Thus lookup does do separate lookups for different exact match identifiers in a common table. Despite this, this can improve hash performance as the free space is, in some sense, shared across all prefix tables of all lengths. The only disadvantage is the use of 33 bit identifiers, which is backward in 32-bit architectures. A solution is to only encode prefixes up to length 31 in 32 bits, and use a standard encoding and a separate hash table for the presumably small number of full 32-bit addresses.

Question 3. Quantifying the benefits of compressing 1-way branches:

• For a unibit trie that does not compress one-way branches, show that the maximum amount of trie nodes can be O(N ·W), where N is the number of prefixes and W is the maximum prefix length. (Hint: generate a trie that uses log2 N levels to generate N nodes, and then hang a long string of N −W nodes from each of the N nodes.)
Solution: In the suggested example, the total number of nodes is clearly N(W − log2 N) which is O(N ·W)

• Show that a unibit trie with text strings to compress one-way branches can have at most 2N trie nodes and 2N text strings.
Solution: The easiest way to see this is to consider inserting a prefix into the tree. After finding where the new entry diverges from the current trie, a new branching node may be added at that point and a new terminal node for the new entry. In the worst case, a text string must be broken up and a new text string added in addition to the branch node. Thus each insertion adds at most 2 trie nodes and 2 text strings, and thus the total across N insertions is bounded by 2N trie nodes and text strings.

Question 5. Reducing Memory References in Lulea: The naive approach to counting bits shown in Figure 11.15 should take three memory references (to access numSet, to read the appropriate chunk of the bit map, and to access the compressed trie node for the actual information.) Show how to use P4a to combine the first two accesses into a single access.
Solution: The simplest solution is to interleave the bitmap and summary count arrays such that the summary count for a chunk is immediately after the bitmap for a chunk. Array indexing for bitmaps and chunks just has to be adjusted slightly to multiply the index by a bigger quantity equal to the sum of the lengths of the elements of the two arrays being interleaved. Once this is done, a simple wide word lookup can retrieve both the appropriate summary count in one memory access, cutting down the total from 3 to 2 memory accesses overall.

Question 11. Invariant for Binary Search on Prefix Lengths: Designing and proving algorithms correct via invariants is a useful technique even in Internet Algorithmics. The standard invariant for ordinary binary search when searching for key K is: "K is not in the table, or K is in the current range R". Standard binary search starts with R equal to the entire table and constantly halves the range while maintaining the invariant. Find a similar invariant for binary search on prefix ranges.
Solution: The invariant for binary search on prefix lengths is similar to binary search but subtly different. It is: "EITHER (The longest matching prefix of D is bmp) OR (There is a longer matching prefix of D in R)", where bmp is the current register holding the current best estimate of the longest match of D. Note that when the range shrinks to zero, it must be the longest match of D is bmp.

Chapter 12: Packet Classification

Question 1. Range to prefix mappings: CAMs require the use of prefix ranges but many rules use general ranges. Describe an algorithm that converts an arbitrary range on say 16 bit port number fields to a logarithmic number of prefix ranges. Describe the prefix ranges produced by the arbitrary but common range > 1024. Given that a rule R with arbitrary range specifications

on port numbers, what is the worst case number of CAM entries required to represent R? Solutions to this problem are discussed in [?, ?].
Solution: Taken from the Fast Layer 4 Switching Paper by Srinivasan, Varghese et al. Suppose we want to convert an arbitrary range X that lies within an enclosing binary range [0, 2k]. Define an anchored range as one that has at least one endpoint at the end of the enclosing range. Then, the arbitrary range X can be split into at most two anchored ranges that lie within [0, 2k−1−1] and [2k−1, 2k] Each anchored range can be split into at most a logarithmic number of prefix ranges by constantly halving the range – at each stage, the halving contributes at most one prefix range. The net result is that we can represent an arbitrary subrange of [0, 2k] with at most 2k prefix ranges. As an example, with 16-bit port numbers the range ≤ 1023 can be expressed using the prefix range 000000*. On the other hand, the range ¿ 1023 can be expressed with 6 prefix ranges 000001*, 00001*, 0001*, 001*, 01*, and 1*.

Question 2. Worst-case storage for set pruning trees: Generalize the example of Figure ?? to K fields to show that storage in set pruning tree approaches can be as bad as O(Nk/k).
Solution: Let N/k = s The first s = N/k entries have the destination fields defined (using s unique destinations) and the remaining fields wildcarded. The next s entries have the source field defined only (using unique sources) and the other fields wildcarded. The next s entries have the third field defined and the other fields wildcarded, and so on. The last s = N/k entries have field k defined and the other fields wildcarded. The net amount of entries caused by replication is (N/k)k = Nk/kk.

Question 3. Grid-of-tries Example Consider the classifier shown in Figure 6. Draw a grid-of-tries structure for this classifier and describe how this structure is navigated when searching for a packet whose first field begins with 000000 and whose second field begins with 101100.
Solution: The upper trie is associated with the first field while the bottom ones are associated with the second field. Consider a search for a highest priority rule that matches a packet whose first field begins with 000000 and whose second field begins with 101100. A longest matching prefix on the first field returns 00 which is associated with the left most node in the first trie. The search continues by traversing the trie linked to the node associated with the prefix 00. The first matching rule which is met is P1 which corresponds to the prefix 1011 in the second field. Trying to continue further the search fails because there is no link from this node that is associated with a value of 0. Also another reason of stopping the search at this point might be noticing that P1 is in fact the highest priority rule in the classifier.

Question 4. Improvements to Grid-of-tries: In the grid-of-tries, the only role played by the Destination Trie is in determining the longest matching destination prefix. Show how to use other lookup techniques to obtain a total search time of (logW + W) for destination-source rules instead of 2W.
Solution: The simple observation is that in grid-of-tries, the first lookup in the destination trie can be made by any lookup algorithm, for example by binary search on prefix lengths in logW time. Unfortunately, there is very little that can be done about the lookup in the source tries with the corresponding switch pointers. This takes W time. Thus the total is logW +W

Question 6. Aggregate Bit Vector Storage: The use of summary bits appears to increase storage. Show, however, a simple modification in which the use of aggregates can reduce storage if the

bit vectors contain large strings of zeroes. Describe the modifications to the search process to achieve this compression. Does it slow down search?
Solution: The main idea, due to Florin Baboescu, is as follows. Whenever a summary word is equal to 0 in a summary bitmap for the field, the entire word can be removed (since the fact that the OR is 0 implies that we know that all bits in the word are 0) from the original field bitmap. This can greatly reduce the storage required for the actual bitmaps of each field entry, allowing large databases to fit into on-chip memory. For example, in the previous example, the bitmap 00000111 with summary of 011 will now be compressed to 00111 (Recall that the third word is only 2 bits, and the first two words are 3 bits each).

The two main complications are that one has to deal with variable size allocation (see Lookups chapter) which can be a pain. Secondly, in accessing a word one cannot use simple indexing. For example, suppose one wants the third in the uncompressed bitmap 00000111. Before compression, this was simply the third word of the bitmap, but now it is the second word in 00111. But as in the Lulea scheme, this indexing is easy to do by counting the number of 1's in the summary bitmap up to and including the position. For example, counting 1's in 011 up to position 3 gives us 2, which is indeed the index into 00111 that we need. Counting 1's is easy in hardware for small sized summaries. Larger size summaries require the Lulea trick of adding precomputed counts of partial chunks.

Question 7. On-demand cross-producting: Consider the database of Figure and imagine a series of web accesses from an internal site to the external network. Suppose the external destinations accessed are D1, . . . ,DM. How many cache iterms will these headers produce in the case of full header caching versus on-demand cross-producting?
Solution: Also taken from the Layer 4 Switching Paper. These addresses correspond to two crossproduct terms (*, Net, * , * , TCP-ACK) and (* , Net, * , * , *). While full-header caching will result in 2M distinct entries in the cache, cross-producting cache will need only two entries. Examples like these suggest that the hit rates for the cross-product cache should be much better than standard header caches.