Route Oracle: Where Have All the Packets Gone?

Yaping Zhu and Jennifer Rexford
Princeton University
yapingz@cs.princeton.edu,
jrex@cs.princeton.edu

Subhabrata Sen and Aman Shaikh
AT&T Labs–Research
sen@research.att.com,
ashaikh@research.att.com

ABSTRACT

Many network-management problems in large backbone networks need the answer to a seemingly simple question: where does a given IP packet, entering the network at a particular place and time, leave the network to continue on its path to the destination? Answering this question at scale and in real time is challenging for several reasons: (i) a destination IP address could match several IP prefixes, (ii) the longest-matching prefix may change over time, (iii) the number of IP prefixes and routing protocol messages is very large, and (iv) network-management applications often require answers to this question for a large number of destination IP addresses in real time. In this paper, we present an efficient algorithm for tracking prefix-match changes for ranges of IP addresses. We then present the design, implementation, and evaluation of the Route Oracle tool that answers queries about routing changes on behalf of network management applications. Our design of Route Oracle includes several performance optimizations, such as pre-processing of BGP update messages, and parallelization of query processing. Experiments with BGP measurement data from a large ISP backbone demonstrate that our system answers queries in real time and at scale.

1. INTRODUCTION

Today, the Internet is the de facto platform for most of our communication needs. Managing and operating networks comprising the Internet is a daunting task especially since the original Internet and protocols underpinning it were not designed with management and operational challenges in mind. Thus, sometimes even answering a basic question like what paths traffic took at a given time is a challenging task. However, knowing the answer to this seemingly simple question is crucial for performing a wide range of network management tasks. In this paper, we focus on answering this question from a given network's point of view: when traffic to a given destination enters the network at some router, what router it will use to exit the network and, beyond that, what downstream path it will take to reach the destination. This paper describes a system called *Route Oracle* to answer this question at scale and in real time.

The Internet is divided into thousands of networks called autonomous systems (ASes), each of which has network elements, such as routers and switches, administrated by a single organization. The Border Gateway Protocol (BGP) allows ASes to exchange information about how to reach external destinations. A BGP route contains many attributes including these key ones: the egress router (i.e., the last router before packets leaves a given AS) and the AS path (i.e., the list of ASes on the downstream path to the destination. Thus, knowing the egress router and AS path for an individual IP address, both current and at a given time in the past, is crucial for several management tasks such as troubleshooting reachability problems in response to customer complaints. While knowing the egress router and AS path already provides vital information, joining this data with other information could form the basis for even more powerful applications. For example, joining the routing data from the intra-domain routing protocol could provide the path traversed by traffic from ingress to egress routers in an AS. Similarly, joining with the performance measurements for an application would allow network operators to correlate performance impairments with route changes in the network.

Realizing a system to track BGP route changes is challenging for two reasons:

- In BGP, routing updates are advertised in terms of prefixes, and prefixes can be nested. Thus a single IP address could be covered by multiple prefixes. The packet forwarding in such cases is governed by the longest matching prefix, which can change over time as routes are advertised and withdrawn. Thus, in order to know the route for a particular IP address, we need to track the changes to its longest matching prefix, and this needs to be done at the rate of BGP updates, especially for applications that need to know routes in real time.

- Some applications need to know routes to hundreds of thousands of IP addresses at a given time. An example is a CDN serving a large number of clients that wants to take into account the current route in deciding which replica to serve the content from. For such applications, our system needs to answer a large number of queries, again in real time.

The Route Oracle system, which outputs route changes for all IPs addresses, overcomes these challenges in the following manner:

- To handle nested prefixes, we introduce the notion of an *address range*. An address range is a group of IP addresses that have the same set of matching prefixes. The algorithm to efficiently determine changes to address ranges as well as their matching prefixes and associated routes as BGP updates arrive. The resulting data structure facilitates rapid determination of route changes for an IP address – we just need to look for the appropriate address ranges instead of prefixes. We also archive the address range data which allows us to determine routes both in present and any time in the past.

- To answer queries for a large number of IP addresses, we implement two optimizations to Route Oracle: (i) the reading of address range records is done once for multiple queries to amortize the I/O cost. (ii) take advantage of multi-core processors prevalent today by distributing the workload across individual cores.

We have deployed Route Oracle in a large Tier-1 ISP where it is being used to troubleshoot performance impairments for various services. In fact, the optimizations described above were implemented after the initial trial which revealed that the users were interested in using the system for hundreds of thousands of IP addresses in real time.

The rest of the paper is organized as follows. In Section 2, we further motivate Route Oracle by describing three applications of the tool. Next, we present the algorithm to efficiently track prefix and route changes for address ranges in Section 3, followed by the overall design and implementation of Route Oracle in Section 4. In Section 5, we evaluate the performance of Route Oracle. We present related work in Section 6, and conclude the paper in Section 7.

2. MOTIVATING APPLICATIONS

In this section, we present three possible applications of Route Oracle. We show that determining the egress router and AS path to external destinations is an important building block for many network management tasks. For each application, we present the challenges of the application, and how Route Oracle could effectively handle it. We also explain why each application needs to track routing changes at scale and in real time.

2.1 Historical Traceroute

Traceroute is a popular active measurement tool, which lists the hops along the path to the destination. It is extensively used by the network operators to troubleshoot reachability or performance problems. Traceroute is implemented by sending IP packets in real time. Therefore, if traceroute was not used or the result was not collected in the past, it would be impossible to roll back to a time in the past to see the path at that time.

However, having a "historical traceroute" that provides paths from any vantage point to a given destination at any time in the past would be valuable. The straightforward approach to implement such historical traceroute is to collect and archive the traceroute results. However, collecting traceroute data from many vantage points to all the destination IP addresses is too expensive due to large probing and storage requirements. Instead, we propose to leverage the passively collected routing updates: from the BGP updates, we could determine the egress router and AS path the packets traverse; and from the routing messages of the intra-domain routing protocol like OSPF, we could determine the hop-by-hop path (from the vantage point to the egress router) inside the ISP. Our Route Oracle tool provides the former capability.

2.2 Analyzing External Routing Disruptions

One of the most important tasks of network operations is to react quickly to large network disruptions. Disruptions like a cable cut could result in loss of reachability or serious congestion in a large portion of a geographical region. Characterization of these events could help network operators understand the impact on their networks: what percent of the IP addresses in the affected geographical region became unreachable? How was the traffic rerouted to reach the destination, and did it cause any traffic shifts inside the network? Did the packets actually reach the destination, and were there any performance degradations? Knowing the answers to these questions would help the network operators make decisions on how to respond to the event, including redirecting the traffic through a better route, or performing traffic engineering to direct traffic away from congested links.

Answering these questions requires determining the routes or route changes of all the IP addresses in the affected geographical regions. Route changes for these IP addresses should not only be processed at scale, but also in real time, in order to get the latest status of the event. Based on these results, aggregated statistics like the percent of routable IP addresses could indicate the overall impact on reachability. Aggregating the routes by the dimensions of egress router and nexthop AS could tell how the traffic shifted inside the network. Joint analysis with the traffic and performance data could help monitor the performance along each rerouted path.

2.3 Service-Level Performance Management

With the deployment of many network applications, ISPs are facing the challenge of *service quality management*. Performance monitoring at the application layer alone is not enough to troubleshoot performance degradation of network services. Instead, network operators need to correlate the performance problems with network-layer changes (such as routing changes, or network congestion) for rootcause analysis.

As an example, consider an ISP hosting a Content Distribution Network (CDN). Such an ISP would be interested in monitoring the performance of its CDN, by measuring the round-trip time (RTT) to the clients. A simple approach is to monitor the performance of content download for each client. However, the RTT of individual clients is likely to vary significantly. Instead, if we could aggregate the RTTs over clients using the same paths (e.g., clients leaving the network at the same egress router), then large changes in the average RTT would show the performance changes caused by rerouting or congestion inside the network. In this case, the capability to determine the routes for all the clients' IP addresses at scale and in real time is required to get the paths from the server to the clients. Furthermore, this real-time performance information could be used to guide the selection of the replica that should serve each client.

3. TRACKING PREFIX MATCH CHANGES

In this section, we present a novel algorithm to determine routes (more specifically egress router and AS path) to destinations given IP addresses, vantage point, and time. We use the BGP routes from the specified vantage point and time period as inputs for the algorithm. Since an IP address could be covered by multiple prefixes, the key problem is to track the longest prefix match and the associated routes for the IP addresses of interest as BGP updates are received.

We begin by describing the phenomena of prefix nesting, which causes an IP address to be covered by multiple prefixes. Then we introduce the notion of an *address range* which is a group of IP addresses that have the same set of matching prefixes, to track routing changes efficiently. Last, we present the algorithm to track prefix match changes as BGP updates arrive.

3.1 Prefix Nesting

In the routing table, an IP address could be covered by multiple prefixes. For example, IP address 128.112.0.0 could be covered by both 128.112.0.0/16 and 128.112.0.0/24. When the IP packets destined to 128.112.0.0 are forwarded, routers perform the longest prefix match (LPM), and use the route of prefix 128.112.0.0/24 to deliver the packets to the destination. In a previous study [1], we analyzed a BGP routing table collected from a router in a large ISP on February 1, 2009. The result showed that 24.2% of the IP addresses were covered by multiple prefixes.

Nesting of prefixes is quite common for a variety of reasons. For example, regional Internet registries allocate large address blocks to Internet Service Providers (ISPs), who in turn allocate smaller blocks to their customers. Customers that connect to the Internet at multiple locations may further sub-divide these address blocks to

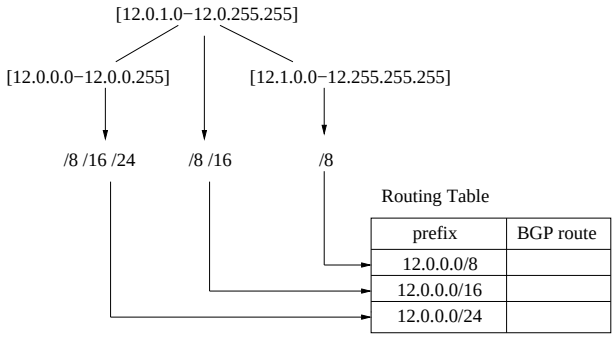

Figure 1: Storing address ranges and prefix sets for prefixes 12/8, 12/16, and 12/24

exert fine-grained control over load balancing and backup routes. ISPs may also announce multiple blocks to protect themselves from route hijacking—for example, AT&T announces prefixes 12.0.0.0/9 and 12.128.0.0/9, in addition to the 12.0.0.0/8 supernet, to prevent other ASes from accidentally hijacking traffic intended for destinations in 12.0.0.0/8.

As a router receives more and more BGP updates, the longest prefix match for an IP address may change over time. In [1], we used BGP update messages collected for the month of February 2009 from the router in the tier-1 ISP backbone, and determined the frequency of BGP updates that affected the longest-matching prefix for IP addresses. Our analysis revealed that 13% of the BGP updates caused some IP addresses to change their longest-matching prefix. Because of frequent changes in longest prefix match, tracking the prefix match changes efficiently is necessary.

3.2 Data Structure for Address Ranges

In this section, we present a method to track prefix match changes for a group of IP addresses. Because of the nesting of prefixes, an IP address could match several prefixes with different mask lengths. In order to track prefix-match changes over time, we need to store information about changes to all prefixes covering the IP address. We refer to the collection of all matching prefixes for a given IP address as its *prefix set*; packet forwarding is driven by the longest-matching prefix in the set at any time. For example, suppose a BGP routing table contains prefixes 12.0.0.0/8 and 12.0.0.0/16. Then, IP address 12.0.0.0 has the prefix set {/8, /16}. IP address 12.0.0.1 also matches the same prefixes. However, the prefix set for 12.1.0.1 is {/8}.

Rather than tracking the prefix set for each individual IP address, we group contiguous addresses that have the same prefix set into an *address range*. For example, prefixes 12.0.0.0/8 and 12.0.0.0/16 divide the IP address space into two address ranges—[12.0.0.0, 12.0.255.255] with prefix set {/8, /16} and [12.1.0.0, 12.255.255.255] with prefix set {/8}. Note that address ranges differ from prefixes in that the boundaries of an address range are not necessarily powers of two. For instance, no single prefix could represent all IP addresses in the range [12.1.0.0, 12.255.255.255].

As we process BGP update messages, address ranges may be created, subdivided or updated. For ease of searching for the affected address range(s), we store information about address ranges in a binary tree, as shown in Figure 1. A binary tree efficiently supports all the operations we need (including inserting a new address range, lookup an address range) in an average time of $O(log\, n)$, where n is the number of address ranges. In comparison, the brute-force solution that simply stores address ranges in arrays would operate in an average time of $O(n)$. Each node in the binary tree contains the left-most address in the address range, and each node keeps a pointer to the size of the address range and the associated prefix set. Each element of the prefix set includes a pointer to the BGP route for that prefix; to save memory, we store a single copy of each BGP route. As illustrated in Figure 1, both address ranges [12.0.0.0, 12.0.0.255] and [12.0.1.0, 12.0.255.255] have prefix 12.0.0.0/16 in their prefix sets, and their prefix sets store the pointers to the route entry for 12.0.0.0/16. Note that in the figure, we only show the pointers from the most-specific prefixes to the routing table for illustration.

3.3 Tracking Changes to Address Ranges

Next, we present an algorithm that reads BGP table dumps or update messages as input, and tracks the changes to the address ranges and their associated prefix sets. The algorithm first determines the address range(s) covered by the prefix, perhaps creating new address ranges or subdividing existing ones. Then, for each of the associated address ranges, the algorithm modifies the prefix set as needed.

Updating address ranges: A BGP announcement for a new prefix may require creating new address ranges or subdividing existing ones. For example, suppose 18.0.0.0/16 is announced for the first time, and no earlier announcements covered any part of the 18.0.0.0/16 address space; then, our algorithm inserts a new address range [18.0.0.0,18.0.255.255], with a prefix set of $\{/16\}$, into the binary tree. As another example, suppose we have previously seen route announcements only for 12.0.0.0/8 and 12.0.0.0/16; then, the binary tree would contain [12.0.0.0,12.0.255.255] with prefix set $\{/8, /16\}$, and [12.1.0.0,12.255.255.255] with prefix set $\{/8\}$. On processing an announcement for 12.0.0.0/24, our algorithm would subdivide [12.0.0.0,12.0.255.255] into two address ranges—one with prefix set $\{/8, /16, /24\}$ and another with $\{/8, /16\}$, as shown in Figure 1. Currently, our algorithm does not delete or merge address ranges after withdrawal messages. We take this lazy approach towards deleting and merging address ranges because withdrawn prefixes are often announced again later, and because we have seen empirically that the number of address ranges increases very slowly over time.

Updating prefix set for address ranges: Continuing with the example in Figure 1, suppose the route for 12.0.0.0/16 is withdrawn. Then, the algorithm would determine that both [12.0.0.0-12.0.0.255] and [12.0.1.0,12.0.255.255] have $/16$ removed from the prefix set. For addresses in [12.0.1.0-12.0.255.255], the withdrawal would change the longest matching prefix to the less specific 12.0.0.0/8.

4. DESIGN AND IMPLEMENTATION

Using the algorithm of Section 3 as a basis, we present the overall design and implementation of Route Oracle in this section. We begin with the overview of the design, and follow it with a detailed description of the two main parts of the design: the preprocessing module that converts BGP updates to updates of address range records, and the query module that identifies the routes for IP addresses based on these records.

4.1 Design Overview

The input to Route Oracle consists of a list of IP addresses, a vantage-point router, and the time period over which route changes are desired. The output is routes for each IP address at the start time, and then changes to them during the specified time period. The route includes the longest matching prefix, egress router and

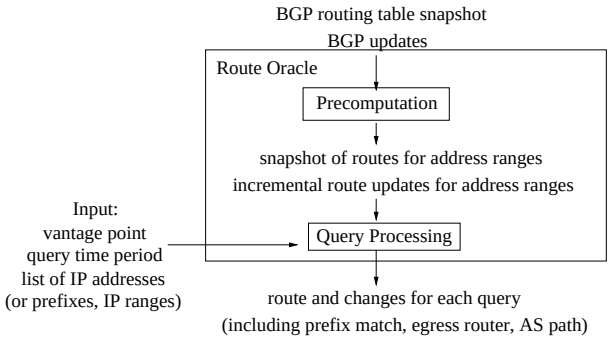

Figure 2: Route Oracle System Design

the AS path. The basic way to handle such a query is to process the BGP updates from the specified vantage point and time period using the algorithm we presented in the last section, and output the result. For example, for the query of route changes for IP address 128.112.0.0 from t1 to t2 at vantage point v, the algorithm will have to process BGP updates from vantage point v from t1 to t2. The output could be: at time t1, the longest prefix match is /24, with associated route r1; at time t3 (which is before t2), the IPs change to a less-specific prefix match /16, with route r2; and finally no route changes for the address till t2. However, this basic approach does not scale well with the number of IP addresses due to nesting of prefixes. The nesting of prefixes means that for each IP address, every update containing a matching prefix needs to be processed to see if the longest prefix match, and hence the route, changed for the address or not.

To overcome this, we convert prefix-based BGP updates to routing changes for address ranges by using the algorithm specified in Section 3. Since address ranges do not overlap, answering queries for route changes scales much better for a large number of addresses in the input. In fact, once we convert BGP updates into address ranges, we save the resulting records for future use, and answer all queries using these records, thereby amortizing the cost of conversion across all subsequent queries.

To summarize, our design of Route Oracle consists of two modules as shown in Figure 2: A pre-processing module that transforms the BGP update stream into a stream of address ranges and associated route changes; and a query module that determines route changes for desired IP addresses using the address range records. In the following subsections, we provide detailed descriptions of these two modules.

4.2 Precomputing Route Updates for Address Ranges

The pre-processing part uses the algorithm presented in the previous section to track the longest prefix match and route changes for all the IP addresses. The inputs to the pre-processing part are BGP routing table snapshots and BGP updates. The output consists of address ranges and route changes to them. Each record contains the following attributes: the address range, time of the route change, the longest prefix match, the egress router, and the AS path. In addition, we also include information on how the longest prefix match was changed compared with the previous route (e.g., change to a more-specific or less-specific prefix), and whether the egress router and AS path were changed compared to the previous route.

Since BGP is an incremental protocol where updates are only sent by a router when its route to a prefix changes, the BGP monitor deployed in the tier-1 ISP generates periodic snapshot of routes for all prefixes. Using the snapshot of prefixes and their routes, the module determines and stores a snapshot of all the address ranges and routes associated with them. It then records changes to address ranges and routes as subsequent BGP updates are received.

We should note that the time interval between snapshots of all address range records has a direct bearing on query processing time since, given a query, we have to process address ranges starting from the latest snapshot before the query time period. Thus, storing snapshots more frequently will reduce query time. However, this will increase the storage space, and hence there is a tradeoff between how often snapshots are stored versus query processing time. At present, we store one snapshot per day to coincide with the BGP routing tables which are generated at the beginning of every day.

We store all the address range records for a given day under the same directory, with the file names indicating the IP address of the vantage point. Directories are named hierarchically according to year, month and day. This facilitates easy searching of files for a given time period. The address range updates are stored in multiple files with each file spanning a fixed interval (currently 15 minutes) of the day. The files are compressed to reduce the storage space. All the records are written in network byte-order for platform compatibility.

4.3 Query Processing

The query module uses the address range records (snapshots + updates) to answer queries regarding routes for IP addresses. The input to the query module is a list of IP addresses, the vantage point and the time interval over which routes are desired. The output is routes at the beginning of the time interval for each address range and changes to them over the interval. The input IP addresses can be specified as address ranges or prefixes.

When a query is received, the module first determines the latest address range snapshot prior to the start time of the interval. It then applies updates of address range records to determine the state at the start time, and then continues processing address range records till the end time to determine route changes. For each address range, the module checks if the range overlaps with the input IP address list, and if so, the record is used to update the route for the affected IP addresses. Note that since the address range files are stored hierarchically based on their time stamps, the module can quickly locate files for a specific time interval. For instance, let us assume that the query is for route changes of a single IP address 128.112.0.0 from October 24, 2009 9:00 a.m. to 5:00 p.m.. In this case, the query module will read the address range snapshot at the beginning of the day on October 24, 2009, and then address range updates till 9:00 a.m. to get the latest route for 128.112.0.0 just at the start time of the query time period. Then, the module will read the address range updates from 9:00 a.m. to 5:00 p.m., and output all route changes for the IP address in question.

We have implemented two optimizations to further reduce the query response time. First, if the user inputs a list of IP addresses in a query, we amortize the cost of reading address range records across all the IP addresses. This is because the reading of the address range files dominates the overall query processing time. As we read address ranges sequentially, we compare each record with all the IP address ranges and prefixes in the list. This way, the reading of the result record files is amortized over all IP addresses.

Second, we parallelize the processing of the address range files given the prevalence of multi-core machines. Since address range updates for fixed time intervals are stored in separate files, we parallelize the reading and processing of these files by submitting mul-

tiple processes each processing one file at a time. The route changes determined by all these processes are gathered at the end, and sorted in chronological order before generating the output.

5. PERFORMANCE EVALUATION

This section presents performance evaluation of the Route Oracle tool. For the pre-processing module, our aim is to verify that it can keep up as BGP updates arrive. For the query module, we evaluate its performance – both response time and resource usage – as a function of the size of the input. We also evaluate the effect of optimizations – parallelization in particular – on the performance of the query module.

Our experiments were run on an standard off-the-shelf SMP server, with two quad-core Xeon X5460 Processors. Each CPU is 3.16 GHz and has 6 MB of cache. The server contains 16 GB of total RAM. We apply our tool to BGP routing table and update messages collected for the month of August 2009 from a top-level route reflector in a tier-1 ISP backbone. The queried IP addresses were randomly selected from the routing table.

We first evaluate the pre-computation module of the tool, focusing on the time to process BGP updates. Next, we evaluate how the query processing time varies as the query time period grows. Then, we evaluate the scalability of the query processing, as the number of queried IP addresses grow, and show the memory resource consumption. Finally, we demonstrate the benefit of parallelization in query processing.

5.1 Pre-processing Time

In this subsection, we evaluate how long it takes to convert BGP updates into address range records. To do this, we consider BGP updates received over fixed time-intervals of 5, 10 and 20 minutes, and compute the time spent by the pre-processing module on each batch of updates. We present results for updates received on August 01, 2009; similar results were observed for other days.

Figure 3 shows the pre-processing time for BGP updates received over various fixed time-intervals as a CCDF (complementary cumulative distribution function). As can be seen, of all the BGP updates received and processed in 5 minutes interval, 99% could be processed within 2 seconds, and the maximum time to process the BGP updates received in 20 minutes interval is about 5 seconds. This clearly demonstrates that the pre-processing module is able to handle BGP updates in real time as they arrive.

In order to better understand what factor mainly determines the pre-computation time, we counted the number of BGP updates received over fixed time intervals. For each point in Figure 4, the number on the x-axis stands for the number of BGP updates received over the fixed time-interval, while the number on the y-axis presents the time spent on processing the updates. The figure clearly illustrates the linear relationship between the pre-computation time and the number of BGP updates received over the interval. The start point of each curve shows that it takes approximately 1.5 seconds to finish the fixed steps of the pre-computation part which includes bootstrapping the data structures used by the pre-processing module, and writing the address range records to files at the end. As the number of BGP updates increases, the pre-processing time increases linearly.

5.2 Query Processing Time

The query processing time is a function of many parameters, including the length of the query time period, the number of IP addresses queried, and the number of processes run in parallel. In this subsection, we evaluate the query processing time along these three dimensions, and understand how these factors affect the query processing time.

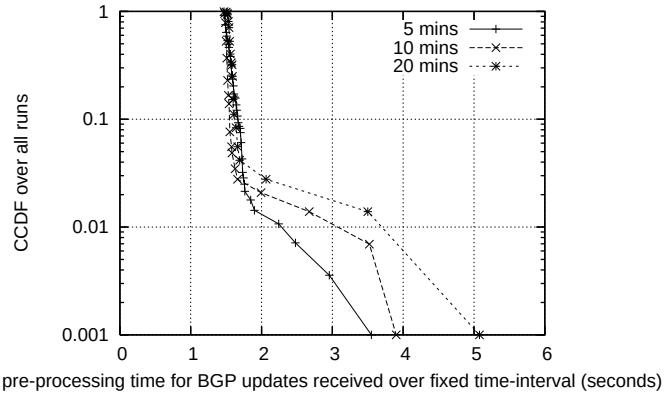

Figure 3: CCDF of the time spent by the pre-processing module to convert BGP updates received over fixed time-interval into address range records.

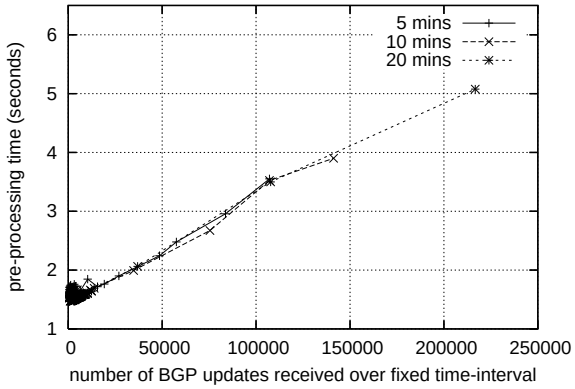

Figure 4: Pre-processing time varies according to the number of BGP updates received over the fixed time-interval.

We start by measuring the query processing time for a single IP address, where a single process is used to handle the query. We vary the start and end times of the query time period on August 01, 2009. We first choose a random start time on that day, then choose a random end time between the start time and the end of the day. Figure 5 shows the result, where each point stands for one experiment with a randomly chosen IP address to query, random start time and random end time. The x-axis denotes the query end time from the beginning of the day.

Figure 5 illustrates that the query processing time grows linearly with the end time of the query. This is because irrespective of the start time, the query module has to process all the address range records from the last snapshot (which happens to be the beginning of the day) till the end time since the address range records from the snapshot till the start time are used to determine the routes used at the start time. This explains why the query processing time depends on the length from the beginning of day to the query end time. In addition, note that the time spent on answering the query for a single IP address for a one-day period is no more than 3.5 seconds.

We should also emphasize that the query time can reduced by storing more frequent snapshots at the cost of more storage space.

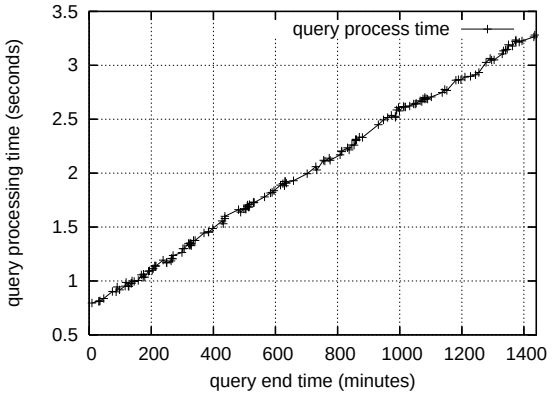

Figure 5: Query processing time versus end time.

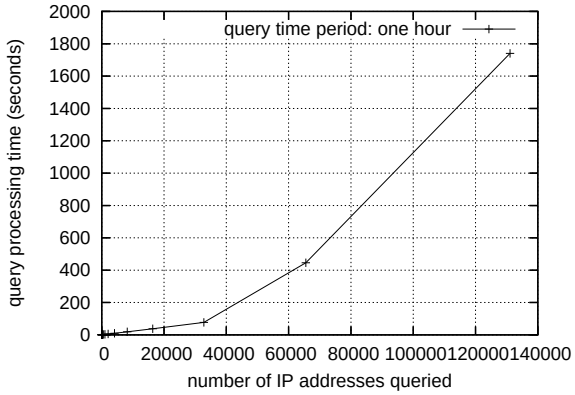

Figure 6: Query processing time versus number of IP addresses queried.

Based on the worst-case time of 3.5 seconds, we believe that storing one snapshot per day provides us with a pretty good performance at a reasonable storage cost (few MBytes per snapshot).

5.3 Scalability of Query Processing

Next, we evaluate the scalability of query processing by increasing the number of queried IP addresses. We also show how the amortization of file processing over the IP addresses reduces the query processing time. In addition, we evaluate the memory consumption as the number of queried IP addresses grows. For this experiment, we use the query time period of one hour. We increase the number of number of IP addresses from 1, 2, 4, 8, ... till about 130,000. The IP addresses used for the queries are all randomly chosen. As in the previous section, we use a single process to answer all the queries.

Figure 6 shows the query processing time in seconds, as the number of queried IP addresses increases. The figure shows that it takes about half an hour to process 130,000 IP addresses for a time interval of one hour. The query processing time grows linearly as the number of queried IP addresses increases. However, the slope is lower at the beginning of the curve, but increases at some point (in fact two points) as the number of IP addresses increases. We believe this is because when number of IP addresses are lower, the time to read the address range records, which is amortized over all IP addresses, dominates. In contrast, when the number of IP addresses becomes large, the time spent on processing the address range records starts dominating, resulting in a higher slope for the curve.

Figure 7 shows the peak virtual memory utilization during the same experiment. As illustrated by the figure, the memory utilization grows as the number of queried IP addresses increases, and the peak virtual memory used for processing about 130,000 IP addresses for an hour is about 480 MB. The increase in memory consumption as the number of queried IP addresses increases is because we store all the matching address range records in memory before outputting them at the end. Another factor affecting the peak virtual memory utilization is the query time period, since the query time period determines the number of address range records that must be stored in memory.

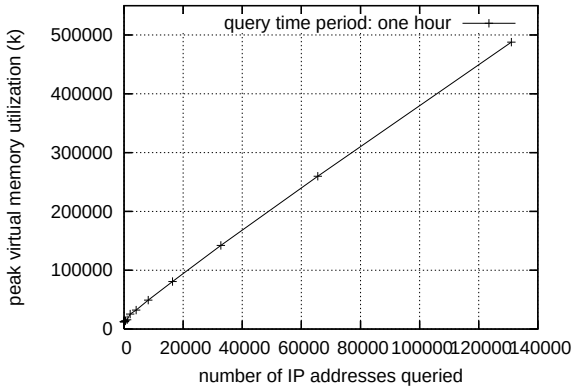

Figure 7: Peak virtual memory utilization versus number of IP addresses queried.

5.4 Parallelization of Query Processing

Last, we evaluate the benefits of parallelization achieved on multi-core machines. For this experiment, we extend the query time period to 10 days from August 01 to August 10, 2009. We vary the number of IP addresses queried by randomly choosing 1, 10, 100, 1000, and 10000 IP addresses. Recall that the server on which experiments were run had two quad-core CPUs, *i.e.*, 8 processing cores each running at 3.16 GHz.

We start by measuring the query processing time by using only *one* process with no parallelization. These numbers serve as a benchmark against which we compare the processing time with multiple parallel processes. Table 1 shows these numbers for varying number of IP addresses over the query time period of 10 days.

Next, for the same set of IP addresses queried, we vary the number of concurrent processes submitted in parallel from 2 to 14. We measure the query processing time, and divide it by the query processing time without parallelization given by Table 1. Figure 8 shows the result of this experiment, where the y-axis is the normalized processing time. As illustrated by the figure, the benefits of parallelization increase, as the number of queried IP addresses increases, since more IP addresses mean more time to process the address range records. For example, the query of 10,000 IP addresses takes about 5 minutes to process without parallelization. This number is reduced to 53%, 30%, 23% and 19% of its value when we increase the parallelization to 2, 4, 6 and 8 processes, respectively. No further gain is achieved in processing time by submitting more than eight processes in parallel.

#IPs Queried	Query Processing Time (Secs)
1	26.1
10	26.1
100	27.7
1000	54.5
10000	279.7

Table 1: Query processing time (without parallelization) versus number of IP addresses queried.

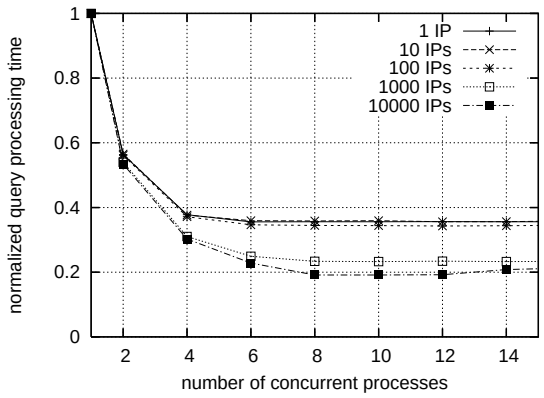

Figure 8: normalized query processing time versus number of concurrent processes

Since the performance levels out when number of processes reach the number of cores, we suspect that the CPU is the performance bottleneck. We confirmed this by tracking the CPU utilization at one minute intervals during the course of the experiment when eight parallel processes were running. As expected, the utilization stayed at 100% during much of the processing, confirming that no further gain is possible beyond eight processes on this particular eight core server.

6. RELATED WORK

Understanding route changes is fundamental to network troubleshooting. Packet Design [2] provides Route Explorer to capture the complete stream of routing updates received. In particular, Route Explorer has an animated historical playback feature which lets the operators diagnose problems by quickly rewinding to past routing activities at a specific time. In our work, Route Oracle takes one step further to handle the prefix-match changes, and outputs route changes for IP addresses of interest.

RouteViews [3] and RIPE-NCC [4] provide publicly available passive measurements of BGP route updates. In these projects, the BGP monitors are fed with BGP announcements and withdrawal messages received via an external BGP session with one router in the participating AS. Proposals have been made to pin-point the location and cause of routing changes using these measurement infrastructures [5, 6, 7]. In comparison, Route Oracle allows one to understand how route changes actually affect routes for IP addresses.

Our work also relates to earlier studies that used BGP measurement data to analyze the relationship between IP prefixes [8, 9, 10, 11, 4]. For example, the work on BGP policy atoms [8, 9] showed that groups of related prefixes often have matching AS paths, even when viewed from multiple vantage points; typically, a more-specific prefix had different AS paths than its corresponding less-specific prefix [8]. Other researchers analyzed BGP table dumps to understand the reasons why each prefix appears in the inter-domain routing system, and the reasons include delegation of address space to customers, multihoming, and load balancing [10, 11]. In contrast, our system focuses on tracking the *changes* in the longest-matching prefix rather than a static analysis of a BGP table dump.

7. CONCLUSION

In this paper, we presented Route Oracle, a scalable system to determine the BGP routes (and thus AS Path and egress router) used by one or more IP addresses from a given router in a network. We believe that the system should form a basis for several network and service management applications. The key component of Route Oracle is an algorithm to track changes to the longest prefix-match for IP addresses as BGP route updates arrive. Our system uses this algorithm to convert BGP updates which are prefix-based into route updates to non-overlapping IP address ranges. This step facilitates queries about route changes for a large number of IP addresses. We also described other optimizations that further reduce the query response time. Our systematic evaluation demonstrated Route Oracle's ability to handle queries at scale and in real time. We have deployed Route Oracle in a large Tier-1 ISP where it is being used to troubleshoot performance impairments for various services. The system optimizations described in the paper were implemented based on feedback from this user community.

8. REFERENCES

[1] Y. Zhu, J. Rexford, S. Sen, and A. Shaikh, "Impact of Prefix-Match Changes on IP Reachability," in *Proc. Internet Measurement Conference*, 2009.

[2] "Packet Design Route Explorer." `http://www.packetdesign.com`.

[3] "University of Oregon Route Views Project." `http://www.routeviews.org`.

[4] RIPE, "RIPE Routing Information Service (RIS)." `http://www.ripe.net/ris`.

[5] M. Caesar, L. Subramanian, and R. H. Katz, "Towards localizing root causes of BGP dynamics," tech. rep., UC Berkeley, 2003.

[6] A. Feldmann, O. Maennel, Z. M. Mao, A. Berger, and B. Maggs, "Locating Internet Routing Instabilities," in *Proc. ACM SIGCOMM*, 2004.

[7] R. Teixeira and J. Rexford, "A Measurement Framework for Pin-pointing Routing Changes," in *ACM SIGCOMM Network Troubleshooting Workshop*, 2004.

[8] A. Broido and k. claffy, "Analysis of RouteViews BGP data: Policy atoms," in *Proc. Network Resource Data Management Workshop*, 2001.

[9] Y. Afek, O. Ben-Shalom, and A. Bremler-Barr, "On the structure and application of BGP policy atoms," in *Proc. Internet Measurement Workshop*, pp. 209–214, 2002.

[10] T. Bu, L. Gao, and D. Towsley, "On characterizing BGP routing table growth," *Computer Networks*, vol. 45, pp. 45–54, May 2004.

[11] X. Meng, Z. Xu, B. Zhang, G. Huston, S. Lu, and L. Zhang, "IPv4 address allocation and BGP routing table evolution," *ACM Computer Communication Review*, January 2005.