SMALTA: Practical and Near-Optimal FIB Aggregation

Zartash Afzal Uzmi[1], Markus Nebel[2], Ahsan Tariq, Sana Jawad
Ruichuan Chen[3], Aman Shaikh[4], Jia Wang[4], Paul Francis[3]
[1]LUMS SSE Pakistan, [2]TU Kaiserslautern, [3]MPI-SWS, [4]AT&T Labs – Research
zartash@alumni.stanford.edu, nebel@cs.uni-kl.de, {ahsan.tariq11, sana.sohail}@gmail.com,
rchen@mpi-sws.org, {ashaikh,jiawang}@research.att.com, francis@mpi-sws.org

ABSTRACT

IP Routers use sophisticated forwarding table (FIB) lookup algorithms that minimize lookup time, storage, and update time. This paper presents SMALTA, a practical, near-optimal FIB aggregation scheme that shrinks forwarding table size without modifying routing semantics or the external behavior of routers, and without requiring changes to FIB lookup algorithms and associated hardware and software. On typical IP routers using the FIB lookup algorithm Tree Bitmap, SMALTA shrinks FIB storage by at least 50%, representing roughly four years of routing table growth at current rates. SMALTA also reduces average lookup time by 25% for a uniform traffic matrix. Besides the benefits this brings to future routers, SMALTA provides a critical easy-to-deploy one-time benefit to the installed base should IPv4 address depletion result in increased routing table growth rate. The effective cost of this improvement is a sub-second delay in inserting updates into the FIB once every few hours. We describe SMALTA, prove its correctness, measure its performance using data from a Tier-1 provider as well as RouteViews. We also describe an implementation in Quagga that demonstrates its ease of implementation.

1. INTRODUCTION

The extreme performance requirements placed on Internet routers lead to difficult engineering constraints. One of these constraints is the size of the forwarding table (or Forwarding Information Base, FIB). One of the factors that goes into determining how big to make the forwarding table is global routing table growth. Router vendors routinely try to set FIB size so as to comfortably accommodate global routing table growth for the expected lifetime of the router. This is not always successful: it is not uncommon for the ISPs to undergo expensive FIB upgrades, or continue to use routers that can no longer hold the full global routing table.

Of course, router vendors also try to use the high-speed memory dedicated to the FIB as efficiently as possible. Considerable effort goes into designing FIB data structures (trees) and corresponding algorithms that result in fast lookup times, fast update times, and are compact in memory [5, 6, 3].

ACM CoNEXT 2011, December 6–9 2011, Tokyo, Japan.
Copyright 2011 ACM 978-1-4503-1041-3/11/0012 ...$10.00.

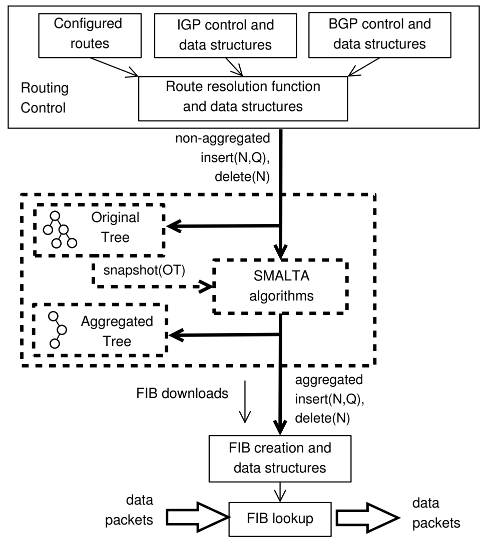
Configured routes
IGP control and data structures
BGP control and data structures
Routing Control
Route resolution function and data structures
non-aggregated insert(N,Q), delete(N)
Original Tree
snapshot(OT)
SMALTA algorithms
Aggregated Tree
FIB downloads
aggregated insert(N,Q), delete(N)
FIB creation and data structures
data packets
FIB lookup
data packets

Figure 1: SMALTA (dashed lines) operates with minimal changes to existing router operation (solid lines). N denotes a Prefix, Q denotes its corresponding Nexthop, and OT denotes Original Tree.

This paper presents a *FIB aggregation* scheme called SMALTA[1] that reduces both the FIB size and the lookup time, thus sharing the goals of FIB lookup algorithms. SMALTA operates without any changes to either the FIB lookup algorithm or the operation of routing protocols and their associated data structures (Routing Information Bases, RIB). Rather, as shown in Figure 1, SMALTA can be inserted between existing router functions, which otherwise can go unchanged.

[1]Saving Memory And Lookup Time via Aggregation.

Original Table	
Prefix	NH
128.16.0.0/15	B
128.18.0.0/15	A
128.16.0.0/16	A

Aggregated Table	
Prefix	NH
128.16.0.0/14	A
128.17.0.0/16	B

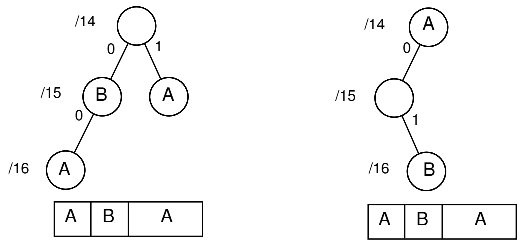

Figure 2: Example of FIB Aggregation (original and aggregated tables and corresponding trees)

In the normal router operation (without SMALTA), the route resolution function takes input from multiple sources (BGP, OSPF, etc.) and computes the set of best-match prefixes and their nexthops (i.e., adjacent routers). This table of best-match prefixes and nexthops is given to the FIB lookup algorithm which generates a tree that is efficient in lookup time, update time, and memory. SMALTA is inserted between these two existing functions. It takes the original table of best-match prefixes and nexthops, and generates a new table that has fewer entries than the first. The new table is semantically equivalent to the original [18]: a lookup on the new table for any address will produce the same nexthop as a lookup on the original table.

Simplistically stated, SMALTA operates by finding pairs of entries that have the same nexthop and can be reduced to a single entry. An obvious example would be two prefixes that are adjacent in the address space. For example, the two prefixes 2.0.0.0/8 and 3.0.0.0/8 could be aggregated to 2.0.0.0/7 if they have the same nexthop. Figure 2 shows a slightly more complex example whereby the two entries with nexthop A can be aggregated to a single /14 entry even though there is an entry with nexthop B in between them.

We are by no means the first to exploit this type of compression. In 1998, Draves et al. designed an algorithm called Optimal Routing Table Constructor (ORTC) that generates a compressed table that is provably optimal by the number of entries in the table [4]. Their algorithm, however, requires that an aggregated table be completely recomputed from scratch with each change to the original table. The cost of this recomputation is linear with the number of nodes in the tree structure, and can take from several hundred milliseconds to a second. It therefore alone is not practical for use with commercial routers.

More recently, Liu et al. [9] designed two algorithms for incrementally modifying the aggregated ORTC table in response to individual updates. These algorithms lack formal proofs (necessary given the complexity of the update process [20]), and are not even fully specified. Besides this, the evaluation in [9] does not take into account aggregation over IGP nexthops (versus BGP nexthops), and it does not measure storage and lookup time improvements (the parameters of actual interest) in the FIB. Indeed we found that the improvement in terms of FIB memory is about 12% less than the improvement in terms of number of entries.

SMALTA is a provably correct incremental FIB aggregation scheme also based on ORTC. In addition to the correctness proof, we automatically computed the correctness of millions of updated aggregated tables (i.e., verified that the use of aggregated table would forward every IP address to the same nexthop as with the original table [18].) We provide a thorough evaluation of SMALTA, using dynamic BGP traces from operational routers as well as for routers with a range of synthetic configurations. Crucially, we focus on the effect to the FIB itself, including memory savings, lookup time, and number and frequency of FIB changes (for the state-of-the-art FIB lookup algorithm Tree Bitmap [5]). We report on our implementation of SMALTA for Quagga [12]. In all, this paper aims to give the ISPs and router vendors an adequate basis for making confident decisions about the pros and cons of implementing and deploying SMALTA. Towards that end, we have presented SMALTA at the IETF [19] drawing significant community interest, and will continue our work to publish SMALTA as an informational RFC.

In all, we find that SMALTA shrinks FIB memory by about one-half. This leads to a roughly 25% improvement in the number of memory accesses for FIB lookup. This also extends the lifetime of the installed router base by roughly four years at current routing table growth. This could be very important should routing table growth accelerate due to the exhaustion of IPv4 addresses. The effective cost of SMALTA is minimal: once every few hours, FIB updates are delayed for around one second or less as the FIB is re-optimized.

This paper makes the following contributions:

- SMALTA, the first incremental FIB aggregation scheme that is: near-optimal, fully-specified, provably correct, and maintains forwarding semantics.

- A thorough evaluation of SMALTA for both deployed routers and synthetically configured routers which, in addition to the traditional measure of number of table entries for BGP nexthops and running time, measures: FIB memory savings and lookup time, the effect of IGP nexthops, the effect of the distribution of prefixes over IGP nexthops, and the effect of SMALTA on the number and burstiness of FIB changes.

- A description of an implementation of SMALTA for Quagga demonstrating the relative ease of introducing SMALTA into a router.

1.1 Outline

Section 2 describes the operation of SMALTA at a high level. Section 3 fully specifies the SMALTA algorithms, and gives an outline of correctness proofs. Section 4 describes our experiments with data from a Tier-1 provider as well as from RouteViews. Section 5 describes our implementation of SMALTA in Quagga. Finally, Sections 6 and 7 present related work, and conclusions and future work respectively.

2. DESCRIPTION

SMALTA operates on a longest-prefix routing table. Table entries consist of an address prefix, denoted as N, and a nexthop, denoted as Q. Packets are forwarded to the nexthop of the longest prefix that matches the packet destination address. Though other data structures may be used, our description, proof, and implementation assume that the routing tables are in the form of a tree, for instance as shown in Figure 2. We will generally refer to a routing table as a tree.

In practice, a routing table is updated as changes to the network or routing policies occur. We specify two functions, *Insert* and *Delete* for these updates. The function *Insert(N,Q)* either inserts a new prefix N into the tree, or changes the nexthop Q of the existing prefix N. The function *Delete(N)* removes an entry from the tree. The term *update* refers to both inserts and deletes.

Figure 1 shows that SMALTA takes as input a stream of non-aggregated updates, and produces a stream of aggregated updates. Internally, SMALTA maintains the non-aggregated tree generated by the received updates, called the *Original Tree* (OT), and the *Aggregated Tree* (AT) produced by the transmitted updates. The result of a single (received) update is zero or more updates transmitted to the FIB. We refer to these transmitted updates as *FIB downloads*. From our measurements (Section 4), SMALTA produces on average slightly less than one FIB download for each update (Figure 10).

The incremental changes to the aggregated tree produced by updates are computationally very efficient. However, these changes do not result in an optimal tree as measured by the number of prefixes stored within the tree. Rather, with each subsequent update, the aggregated tree on average drifts further from optimal.

To remedy this, SMALTA has an internal function called *snapshot(OT)* that takes as input the entire original tree OT and produces a complete new aggregated tree. Like [9], *snapshot(OT)* uses ORTC [4], and as such the aggregated tree it produces is optimal. *snapshot(OT)* is called when the original tree is first initialized, for instance upon router startup. Subsequently, incremental updates cause the aggregated tree to drift away from optimal. *snapshot(OT)* is periodically repeated, for instance after some number of updates, or after the aggregated tree has grown by more than a certain amount. Our measurements show that the aggregated tree drifts only a few percent from optimal even after tens of thousands of incremental updates (Figure 9). With each *snapshot(OT)*, the aggregated tree returns to optimal.

When a router first boots up, it obtains and installs routes from its neighbors before it advertises routes to its neighbors. This way, it does not receive data packets before it is ready to forward them. BGP has an explicit mechanism, the End-of-RIB marker, that allows a BGP speaker to tell its neighbor when its entire RIB has been conveyed [15]. While BGP is initializing but before the End-of-RIB is received, SMALTA inserts updates into the original tree, but does not process them further. In other words, nothing is put into the aggregated tree, and no FIB downloads are produced. After the BGP control has received all End-of-RIB markers from all neighbors, SMALTA runs its initial *snapshot(OT)*.

When this initial *snapshot(OT)* runs, its output is a set of FIB downloads in the form of inserts corresponding to the complete aggregated tree (AT). When *snapshot(OT)* runs subsequently, its output is a set of FIB downloads consisting of both inserts and deletes that comprise the delta between the pre-*snapshot* AT and the post-*snapshot* AT. Specifically, for each prefix N that is in the pre-*snapshot* AT but not in the post-*snapshot* AT, a *Delete(N)* is FIB downloaded. For each prefix N that is not in the pre-*snapshot* AT but in the post-*snapshot* AT, an *Insert(N,Q)* is FIB downloaded. For each prefix N that is in both AT's, but with different nexthop Q, a *Delete(N)* followed by an *Insert(N,Q)* are FIB downloaded. Note that this is essentially what is done today in the context of Graceful Restart [10, 15]. Our measurements show that a call to *snapshot* after 20K updates (about two hours of updates) generates roughly 2000 FIB downloads (Figure 10).

The *snapshot* takes less than a second to run (for current BGP table sizes and on hardware with capabilities comparable to current routers). While the *snapshot* is running, updates are queued up. After the *snaphot* has finished, and the delta FIB downloads are produced, the queued updates are processed. This means that during calls to *snapshot*, a small number of routing events are delayed by a fraction of a second. Practically speaking, assuming that the *snapshot* is called every other hour or so, one in a few thousand routing events will take slightly longer to load into the FIB.

2.1 SMALTA Algorithms

SMALTA uses ORTC for its *snapshot* algorithm [4]. ORTC makes three passes over the tree:

1. **Pass-1:** a normalization pass in which the prefixes are expanded such that every node in the binary tree has two or no children,

2. **Pass-2:** a post-order traversal up the tree, wherein each node is assigned a set of nexthops, and

3. **Pass-3:** in which the algorithm assigns nexthops to prefix nodes in the tree starting from the root and traversing through to the leaves, removing any unnecessary leaves.

2.1.1 The Update Algorithms

Incorporating an update into the AT may at first glance appear simple: a prefix node is either created, updated or its nexthop is removed. Such 'naive' incorporation of the updates into the AT, leads to semantic incorrectness. This is best illustrated by an example. Figure 3 continues the example from Figure 2, additionally showing an insert, with nexthop Q, received for the indicated node. If this update is 'naively' incorporated in AT and OT, the two resulting trees (bottom part of Figure 3) become semantically different.

This example illustrates just one of numerous 'corner' cases that can be encountered; all such cases are exhaustively dealt with in the design of SMALTA algorithms given in Section 3. The incorrectness of direct incorporation of updates into the AT inherently stems from the process of aggregation. To incorporate an update into the AT, we make the following key observation:

The prefix in each BGP update message covers some IP address space. In the original table, a single prefix node covers this space. But, in the aggregated table, it may take multiple nodes to cover the same space.

The SMALTA update algorithms work by identifying the set of such nodes in the AT that cover the same IP space as covered by the prefix in a BGP update message, and then making corresponding changes to those nodes. For the above example, Figure 4 shows the correct way of incorporating the insert.

Next, we present an intuitive but incomplete description of the SMALTA update algorithms. Section 3 provides the complete algorithms and the outlines of their proofs which are detailed in a technical report [20].

Insert(N,Q): In order to insert a prefix N, the address space of N should be assigned to the newly inserted nexthop Q. This is achieved by setting Q as the nexthop of N in the aggregated tree AT (Step-1 in example of Figure 4). This may incorrectly claim extra space covered by some prefixes in OT (specifics of N) that were aggregated into its immediate ancestor prefix in the AT. We must restore all such prefixes (Step-2 in Figure 4). Finally, we must also reclaim the space covered by the specifics (also called deaggregates) of N in favor of Q (Step-3 in Figure 4).

Delete(N): To delete a prefix N, the ownership of the address space originally covered by N should be given up in favor of the nexthop of its immediate ancestor prefix N_O in the OT. This can be achieved by setting nexthop of N in the AT equal to that of N_O in the OT. As before, this may incorrectly claim extra space covered by some prefixes in OT (specifics of N) that were aggregated into its immediate ancestor prefix N_A in the AT. Thus, we restore all nearest descendent prefixes of N if their nexthop does not match the new nexthop of N. Finally, we reclaim the space covered by the deaggregates of N by setting their nexthops equal to that of N_O. Next section provides the details.

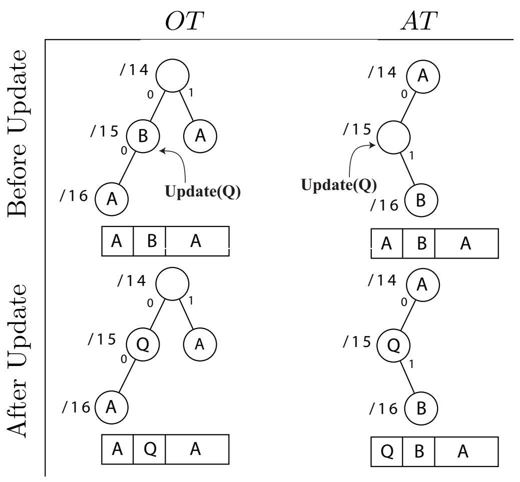

Figure 3: Incorrect way of incorporating an update into the AT. Showing OT (top-left) and corresponding AT (top-right) before the insert is incorporated. Naive incorporation of the update results in correct OT (bottom-left) but incorrect, and semantically different AT (bottom-right).

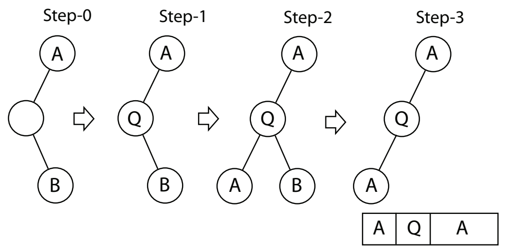

Figure 4: Correct way, with intermediate steps, of incorporating the insert into the AT of Figure 3.

3. SMALTA UPDATE ALGORITHMS

3.1 Definitions

DEFINITION 1. *We use $\{0,1\}^W$ to denote the set of all strings of length W (32 for IPv4) over binary alphabet $\Sigma = \{0,1\}$. The set of prefixes to all these strings, including the empty prefix δ, is denoted by $\{0,1\}^{\leq W}$.*

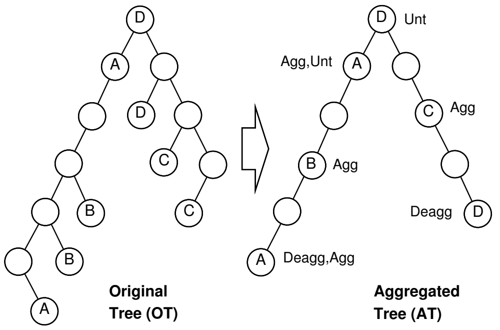

Figure 5: A prefix in AT can be one or some combination of the three types: Untouced (Unt), Aggregated (Agg) or Deaggregated (Deagg) w.r.t. the prefixes in OT. All valid combinations are shown here.

Definition 2. *Let* $\mathcal{P} \subseteq \{0,1\}^{\leq W}$ *and* $\mathcal{N}$ *be the set of all possible nexthops including the null nexthop* ε*. A* prefix table for $\mathcal{P}$ *is given by a total mapping* $d : \mathcal{P} \to \mathcal{N}$*. From* d*, we construct a labeled binary tree in which a node* $n(p)$ *represents* $p \in \mathcal{P}$ *such that* p *describes a path to node* $n(p)$*, i.e., an ith digit* 0 *(resp.* 1*) in* p *corresponds to a left (resp. right) successor on level* i *of that tree. We regard* $d(p)$ *as a label of* $n(p)$ *assuming a labeling of* ε *for every node whose corresponding prefix does not exist in* $\mathcal{P}$*. We use a 'prefix table' and its tree representation interchangeably.*

We use OT (Original tree/table) to denote the table containing prefix set $\mathcal{P}$. The nexthop for a prefix $p \in \mathcal{P}$ is denoted by $d_O(p)$. The function call *snapshot(OT)* results in a table—written as AT (Aggregated tree/table)—which contains a new set of prefixes $\mathcal{P}'$. We use $d_A(p')$ to denote the nexthop for a $p' \in \mathcal{P}'$. Each $p' \in \mathcal{P}'$ can be one or more of three types:

1. An *Aggregate (A)* obtained by shortening (from the right) one or more prefixes in $\mathcal{P}$. The node $n(p')$ is also regarded as an aggregate.
2. A *Deaggregate (D)* obtained by extending (to the right) a prefix in $\mathcal{P}$. The node $n(p')$ is also regarded as a deaggregate.
3. *Untouched (U)* in which case p' is obtained from a $p \in \mathcal{P}$ without alteration.

The example in Figure 5 shows an OT and the resulting AT after calling *snapshot(OT)*, and depicting all the possible combinations of node/prefix types in AT. Thus, each $p' \in \mathcal{P}'$ is generated by (shortening, extending, or keeping intact) some prefixes in $\mathcal{P}$ all of which are called the *preimages* of p'. It may be noted that if $p \in \mathcal{P}$ is a preimage of $p' \in \mathcal{P}'$, then $d_O(p) = d_A(p')$.

Definition 3. *For any two prefixes* p *and* p'*, we write* $p < p'$ *to indicate that* p *is a proper prefix of* p'*; we also write* $p \leq p'$ *to indicate a case where* p *might also be equal to* p'*.*

Definition 4. *For a prefix* p *in* $\circ T$*, where* $\circ \in \{A, O\}$*,*

- $\Psi_\circ(p) :=$ *longest* p' *with* $p' < p \wedge d_\circ(p') \neq \varepsilon$*;*
- $\Psi_\circ^=(p) :=$ *longest* p' *with* $p' \leq p \wedge d_\circ(p') \neq \varepsilon$*;*
- $\Lambda_\circ(p) := \{p' \in \{0,1\}^{\leq W} \mid p \leq p' \wedge (\nexists p'') (p < p'' \leq p' \wedge d_\circ(p'') \neq \varepsilon)\}$*.*

Thus, p with $d_\circ(p) \neq \varepsilon$ is the longest matching prefix for every prefix in the set $\Lambda_\circ(p)$. In other words, $\Lambda_\circ(p)$ is the set of all prefixes p' for which the node $n(p)$ (in tree $\circ T$) is the last explicitly labeled node visited when processing a query for p'. Such a query is answered by traversing $\circ T$ according to the bits in p' until either a leaf is reached or the requested successor of the actual node does not exist. Accordingly, we think of the non-null nexthop of node η to be *propagated* to all its descendants until either a leaf or another explicitly labeled node is reached. The other way round, we will say that node η' *inherits* a nexthop when η' is among the nodes to which a nexthop is propagated.

Definition 5. *A nexthop* h *is said to be* ***present*** *at a node* η *when* η *is either explicitly labeled with* h *(and propagates* h *to all its descendants) or it inherits* h *from a nearest ancestor which propagates* h *to* η*.*

3.2 Algorithms

Once a call to the *snapshot(OT)* results in an AT, subsequent updates are individually incorporated into that AT by calling one of the SMALTA update algorithms. For a prefix N in AT, (i) a new nexthop Q is inserted by making an *Insert(N,Q)* call, and (ii) a prefix N is deleted by making a *Delete(N)* call. In case of a delete, we assume that $d_O(N) \neq \varepsilon$.

We will use index O (resp. O') to address OT before (resp. after) calling one of the above two update algorithms. The notation $pi(N)$ will be used to represent a pointer to the preimage of N in OT; we will also use $\hat{}\eta$ to denote a pointer to node η. For both algorithms, and for each update, variable X (resp. R) is used to store the nexthop *present* at $n(N)$ in AT before (resp. after) that update.

In the Delete algorithm, we use the boolean N^{agg} which is set true if and only if $n(N)$ in AT is a *pure* aggregate i.e. no deaggregate or untouched prefix is associated with $n(N)$.

From the intuitive description of the algorithms in Section 2.1.1, incorporating an update into the AT requires steps to 'repair' the AT ensuring semantic equivalence with the corresponding OT. Algorithms 1 and 2 provide a complete listing of the intuitive explanation given in Section 2.1.1. Both of the SMALTA update algorithms use another algorithm namely reclaim (shown as Algorithm 3) to make some of these repairs.

Algorithm 1 ***Insert(N,Q)***

```
(1)  proc Insert($N, Q$) $\equiv$
(2)    $P := \Psi^{=}_{O}(N)$; $I := \Psi_A(N)$; $pi(N) =$ nil;
(3)    if $d_A(N) = \varepsilon$
(4)      comment: $n(N) \notin \{\text{Agg, Unt, Deagg}\}$
(5)      then if $d_A(I) \neq Q$
(6)             then $X := d_A(I)$; $R = Q$;
(7)                  $d_A(N) := Q$; reclaim($N, R, X$);
(8)           fi
(9)      else comment: $d_A(N) \neq \varepsilon$
(10)          if $d_O(N) = \varepsilon$ or $d_O(N) = d_A(N)$
(11)            then
(12)                 $X := d_A(N)$; $R := Q$;
(13)                 if $d_A(I) = Q$
(14)                   then $d_A(N) := \varepsilon$;
(15)                   else $d_A(N) := Q$; fi
(16)                 reclaim($N, R, X$);
(17)          fi
(18)   fi
(19)   comment: Visit Deaggs of $P$ under $n(N)$
(20)   for $E \in \{ D \mid D$ has preimage $P, N \leq D \}$ do
(21)       $pi(E) := \hat{}n(N)$; $d_A(E) := Q$;
(22)       reclaim($E, Q, d_O(P)$);
(23)   od
(24) end
```

Algorithm 2 ***Delete(N)***

```
(1)  proc Delete($N$) $\equiv$
(2)    $N^{agg}$ := false; $P := \Psi_{O'}(N)$; $I := \Psi_A(N)$;
(3)    if $d_A(N) \neq \varepsilon$
(4)      then if $d_A(N) = d_O(N)$
(5)             then $X := d_A(N)$; $R := d_A(I)$;
(6)                  $d_A(N) := \varepsilon$;
(7)             else $N^{agg}$ := true;
(8)           fi
(9)      else comment: $N$ has been aggregated to I
(10)          $X := d_A(I)$;
(11)   fi
(12)   if not $N^{agg}$
(13)     then if $d_{O'}(P) \neq d_A(I)$
(14)          then $d_A(N) := d_{O'}(P)$;
(15)               $R := d_{O'}(P)$; $pi(N) := \hat{}n(P)$;
(16)           else if $P < I$
(17)               then $R := d_{O'}(P)$; $pi(I) := \hat{}n(P)$;
(18)               fi
(19)          fi
(20)          if $d_{O'}(P) \neq X$ then reclaim($N, R, X$); fi
(21)   fi
(22)   for $E \in \{ D \mid D$ deaggregate of $N \}$ do
(23)       $pi(E) := \hat{}n(P)$; $d_A(E) := d_{O'}(P)$;
(24)       reclaim($E, d_{O'}(P), d_O(N)$);
(25)   od
(26) end
```

3.3 Correctness Proofs: Outline

The basic idea in proving the correctness of our algorithms is as follows:

As a first step, we come up with two invariants which characterize the relationship of aggregates (resp. deaggregates) to their preimages, together with the properties of their connecting paths in the AT (resp. OT).

Algorithm 3 **reclaim**(E,α,β)

For node $n(E)$, α (resp. β) is the nexthop present after (resp. before) a change to AT, i.e. either $d_{A'}(E) = \alpha$ (resp. $d_A(E) = \beta$) or α (resp. β) is propagated up to node $n(E)$ by a predecessor. Then procedure reclaim first optimizes AT making use of the new value α and then reclaims prefixes for nexthop β which were formerly aggregated up and are now incorrectly covered by α.

```
(1)  proc reclaim($E, \alpha, \beta$) $\equiv$
(2) for $n(D)$ a descendant of $n(E)$ in $OT$ or $AT$ do
(3)   if $d_A(D) = \varepsilon = d_{O'}(D)$
(4)     then reclaim($D, \alpha, \beta$);
(5)      else if ($d_A(D) = \alpha$) or ($d_{O'}(D) = \alpha$)
(6)            then if $d_A(D) = \alpha$
(7)                   then $d_A(D) := \varepsilon$; $pi(D) :=$ nil;
(8)                    else if $d_A(D) = \varepsilon$
(9)                           then reclaim($D, \alpha, \beta$);
(10)                        fi
(11)                 fi
(12)            else if ($d_{O'}(D) = \beta$) and ($d_A(D) = \varepsilon$)
(13)                   then $d_A(D) := \beta$;
(14)                    else if ($d_{O'}(D) \neq \beta$) and ($d_A(D) = \varepsilon$)
(15)                           then reclaim($D, \alpha, \beta$);
(16)                        fi
(17)                 fi
(18)          fi
(19)   fi
(20)  od
(21) end
```

We state these invariants here:

Invariant 1. *On the path from a deaggregate p to its preimage p' only null nexthops can be found in OT.*

Consider the example in figure 5. The left child of the root node in OT is the preimage p of the far left node p' (with label A) in AT. Clearly, all nodes between p and p' in OT have null nexthops.

Invariant 2. *On the path from an aggregate p to any of its preimages p' in OT only null nexthops can be found in AT (or the corresponding nodes do not exist in AT).*

Once again, we notice that each aggregated node in the AT of figure 5 exemplifies this invariant.

We then use the construction of AT by ORTC to prove that the invariants are initially fulfilled – i.e., after a call to *snapshot(OT)* [20].

In the second step towards proving our update algorithms (Insert and Delete), we show that a call to either of these algorithms does not affect the invariants. Furthermore, we prove that for any path along which the nexthops (explicit or propagated/inherited) may be affected by a call to these algorithms, the nexthop finally used to answer a lookup query is, in all cases, the same for AT and OT. To show this, we first note that a call to either of our update algorithms for a prefix p can only affect queries for $p' \in \Lambda_\circ(p)$. For those p', the nexthop propagated by p is the one that determines the routing. Thus, assuming that AT has provided the right nexthop for every $p' \in \{0,1\}^W$ before a call to Insert (resp. Delete), it is sufficient to show that within AT the correct nexthop is propagated (resp. inherited) by p after the call to Insert (resp. Delete) and that this call does not cause any side effects for prefixes in $\Lambda_A(\bar{p})$, $\bar{p} \neq p$, in order to prove its correctness.

In summary, the second step shows that after applying one of our update algorithms (i) both trees (OT and AT) remain semantically equivalent, and (ii) both the invariants remain fulfilled.

In the third and final step, we use the proof arguments iteratively to show that any sequence of calls to Insert or to Delete operations is correctly handled by SMALTA. Complete proofs with details of various cases can be found in [20].

Our case selection in the second step is exhaustive as we consider calls to Insert and Delete for all possible node states. In fact, using this generic and exhaustive formulation of cases, we were able to identify some rarely-occurring corner cases, comprising specific sequences of updates, that were undetected in the implementation.

4. PERFORMANCE EVALUATION

Our evaluation of SMALTA spans two data sets, one obtained from routers in a large Tier-1 service provider and another obtained from routeviews [14]. The evaluation criteria focuses on: (i) savings in FIB storage, (ii) reduction in FIB lookup memory accesses, and (iii) changes and delays to FIB updates.

We report SMALTA improvements compared to the case when no FIB aggregation is used. For completeness, we also provide a head-to-head comparison between SMALTA and two previously proposed FIB aggregation approaches, namely Level-1 (L1) and Level-2 (L2) [22, 21]. Similar to how prefix aggregation is done in BGP today, L1 drops more specific prefixes when a less specific prefix has the same nexthop, and L2 additionally aggregates sibling prefixes having the same nexthop. As expected, SMALTA significantly outperformed these simple FIB aggregation approaches in both the amount of FIB memory and the lookup time.

4.1 Data Sets

4.1.1 Tier-1 Provider data set

We gathered routing data for two types of border routers in the Tier-1 provider: those connecting to customer networks (called Access Router, or AR), and those connecting to peer networks (called Internet Gateway Router, or IGR). We gathered iBGP updates for the IGRs, which acted as route reflectors to a monitor similar to [11]. These updates are the result of the best-path selection process by the IGR, and so represent the updates that go to the FIB. We start our traces at BGP reset events. This allows us to construct the initial RIB state from the updates in the burst following the reset event. Subsequent updates from the IGR are regarded as part of the update trace for that IGR. We selected two such IGRs for our evaluation of SMALTA, one on the east coast and one on the west coast of the continental United States. The update trace spans 12 hours. The results for these two IGRs were roughly the same, so we report only on the west coast IGR.

For the ARs, we obtained snapshots of the set of table entries in the FIB. We collected these snapshots for five ARs each with a varying number of IGP nexthops, and use these to report on the effect of IGP nexthops. Note that, in this provider network, neither route reflectors nor core routers are suitable candidates for study. The route reflectors are not in the data plane and so don't require FIBs as such, and the core routers forward MPLS packets and don't contain global routing tables.

4.1.2 Routeviews data set

We used the routeviews [14] BGP feeds to mimic a router with a number of eBGP peers, one per routeviews feed. To do this, we assumed a simple best-path selection policy to determine the updates to the FIB. We also modeled a varying number of IGP nexthops by mapping each eBGP peer to an IGP nexthop in a round-robin fashion.

We generated FIB snapshots from the first RIB data file on December 15 for each year from 2001 to 2010, and then used a full day of subsequent updates.

4.2 FIB lookup algorithm

The percent savings in the number of entries in the Aggregated Tree (AT) does not necessarily reflect the percent savings in FIB memory. This is because FIB lookup data structures employ their own storage optimizations, so the percentage savings in FIB memory may not match the savings in the number of FIB entries. Different FIB lookup algorithms may have different savings. In addition, the reduction in the number of memory accesses for FIB lookup may vary depending on the specific FIB lookup algorithm. In our experiments, we used Tree Bitmap (TBM) as the FIB lookup algorithm [5]. TBM is a state-of-the-art storage-efficient lookup algorithm used in some present-day high-end commercial routers. In particular, we use the software reference design (Sec. 6 of [5]) of TBM to compute FIB storage and memory accesses. We tested a variety of stride lengths and selected the one that minimizes the memory requirement. For our implementation of TBM, we used 32-bit pointers, the Initial Array Optimization followed by a constant stride length of 4. Altogether,

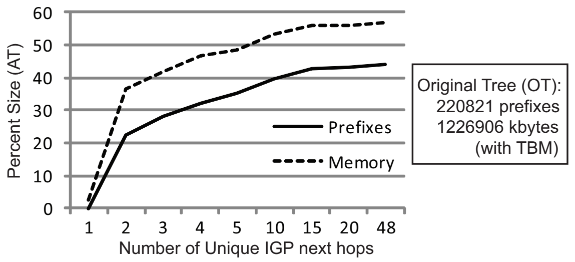

Figure 6: Results (Routeviews, 2006) showing AT size as a percentage of OT size, as a function of unique IGP nexthops (horizontal axis). Results for other years (2001 to 2010) are similar.

the size of a single TBM node in our experiments is 8 bytes. For our experiments, we measure the memory consumed by TBM, as well as the lookup time expressed as the average number of memory accesses per lookup assuming every IP address in the covered space is equally likely to be looked up.

4.3 Results

Effect of the number of IGP nexthops: Previous studies of aggregation have used BGP nexthops. SMALTA, however, aggregates prefixes based on their IGP nexthops.[2] Intuitively, we would expect that one gets better aggregation by using the IGP nexthops instead of BGP nexthops. This is because multiple BGP nexthops may map to a single IGP nexthop, creating additional opportunities to aggregate the prefixes. By the same token, one may expect to achieve higher degree of aggregation if the number of unique IGP nexthops is smaller. To support this intuition, we present two results, one from the routeviews data set and the other from the ARs in the provider data set.

Figure 6 shows the effect of varying the number of IGP nexthops from one to 48, the total number of BGP nexthops for the routeviews collection in 2006. When there is a single IGP nexthop, the snapshot produces an aggregated table with only a single entry. Two IGP nexthops gets slightly under 80% reduction (solid line), and by the time we reach 48 IGP nexthops we get about 55% improvement.

Measurements from the Tier-1 provider show similar trends. Table 1 summarizes the *snapshot* results for five ARs in the Provider network. From the results in this table, however, we see no relation between the number of IGP nexthops and the percent reduction in the number of entries. In particular, AR-1 offers significantly higher reduction (87%) in the number of prefixes; our investigation indicated that this was because most of the prefixes handled by AR-1 were assigned to a single nexthop, some were assigned to a second nexthop and, each of the remaining nexthops provided reachability to only a couple of prefixes each. In other words, the *effective* number of nexthops is small even though the actual number is quite large. To approximately capture this effect for a given AR, we compute the entropy underlying the number of prefixes assigned to each nexthop. From this, we compute the *effective* number of IGP nexthops $E(\mathrm{R})$ for a router R as follows: If f is the total number of unique nexthops on R and n_i is the number of prefixes assigned to the i^{th} nexthop, then:

$$\log_2 E(\mathrm{R}) = \sum_{i=1}^{f} -p_i \log_2 p_i;$$

$$\text{where} \qquad p_i = \frac{n_i}{\sum_{j=1}^{f} n_j}$$

[2]All links in the provider network we used in our experiments were point to point. For such links, IGP nexthops have a one-to-one relation with interfaces.

	AR-1	AR-2	AR-3	AR-4	AR-5
$E(\cdot)$	1.061	1.766	1.845	2.01	3.164
#NH	89	419	25	9	652
$\#(OT)$	427,205	426,175	426,736	427,520	428,766
$\mathrm{T}(OT)$	2.10	2.10	2.10	2.10	2.10
$\#(AT)$	56,486	81,456	91,039	171,996	237,915
$\mathrm{T}(AT)$	1.09	1.18	1.26	1.53	1.68
#(L1)	209,686	231,129	235,532	292,432	340,571
T(L1)	1.89	1.92	1.92	1.99	2.02
#(L2)	118,980	147,376	158,925	248,295	307,442
T(L2)	1.78	1.82	1.85	1.96	2.02

Table 1: Impact of the actual number of IGP nexthops #NH and *effective* number of nexthops $E(\cdot)$ on the aggregation results for ARs after applying *snapshot*. $\mathrm{T}(\cdot)$ refers to the number of lookup memory accesses.

Table 1 also lists the number of effective nexthops (first row) for each of the five service provider ARs and we notice their correlation with the percent reduction in the size of the AT. This is visually depicted in Figure 7. The trend in this figure (shown as dotted) corroborates with the graphs we see from the routeviews data (Figure 6), i.e., increasing the number of distinct effective nexthops reduces the extent of aggregation.

In addition to SMALTA results (depicted by $\#(AT$ and $\mathrm{T}(AT))$, performance results for L1 and L2 approaches are also shown in Table 1. Clearly, SMALTA achieves significantly better aggregation. Furthermore, while L1 and L2 bring about a moderate reduction in the lookup time (depicted by the parameter $\mathrm{T}(\cdot)$), we note that SMALTA results in much faster lookup speeds. These results remained consistent across all our experiments in all data sets.

TBM efficiency: Figure 6, as well as Table 2, show that the memory savings of TBM, while substantial, are not as good as the savings in the number of prefix entries in the aggregated tree: roughly 12% less savings overall. In Figure 6, the savings difference is consistent from 2

	Initial Snapshot	After 183719 Updates (12 hrs)
#(OT)	418,033	418,090
M(OT)	2,361,714	2,362,460
T(OT)	2.103	2.104
#(AT)	156,877 (37.5%)	159,866 (38.24%)
M(AT)	1,177,138 (49.84%)	1,188,188 (50.29%)
T(AT)	1.550 (73.7%)	1.553 (73.8%)
#(L1)	282,641 (67.6%)	-
M(L1)	1,673,242 (70.85%)	-
T(L1)	1.974 (93.8%)	-
#(L2)	219,704 (52.6%)	-
M(L2)	1,486,144 (62.93%)	-
T(L2)	1.927 (91.6%)	-

Table 2: IGR-1 (8 IGP nexthops) aggregation before and after the updates. M(·) denotes the FIB memory (bytes), and T(·) the average number of memory accesses for a lookup. IGR-2 produced similar results.

IGP nexthops and up. The case of a single IGP nexthop represents a non-linearity (all prefixes aggregated to a single entry) and is not expected to occur in practice.

Note that FIB data structures other than TBM may experience different levels of memory savings, depending on the actual mechanism used in storing the FIB entries. Router vendors must test against their own FIB storage methods to determine the benefits of SMALTA.

Table 2 shows that aggregation via SMALTA results in about 25% average savings in number of memory accesses (and hence, lookup time). This assumes a uniform traffic matrix. Similar savings were obtained for provider ARs, as indicated in Table 1 (and Figure 7), though the amount of savings in lookup time varied from 48% (for the router which results in smallest number of entries in AT) to about 20% (for the router with most number of entries in the AT). Experiments with routeviews data showed similar trends. Indeed, when the number of IGP nexthops is one, we just have a single 'Initial Array' for TBM, which results in a single memory access.

These significant savings in memory and lookup time are the primary benefits of SMALTA.

Table 2 also includes the comparative results for the L1 and L2 aggregation approaches. Once again, we note that L1 consistently resulted in roughly 30% less aggregation, and L2 about 15% less. In all cases, the 'end-to-end' FIB memory savings are roughly 12% less than the savings in the number of entries. The savings in lookup times for L1 and L2 are even less prominent, averaging less than 10% compared to the average lookup time reduction of 25% in case of SMALTA.

Given the substantially poorer snapshot performance of L1 and L2 compared to SMALTA, we felt it unecessary to implement and test the performance of incremental updates for L1 and L2. We therefore don't show performance numbers for after the 12 hours of updates for L1 and L2 in Table 2. Nevertheless, as with SMALTA and other update algorithms for instance from [22], we can expect less aggregation with subsequent updates, as shown by Figures 9 and 10 in [22].

Note that the time to process each incremental update is insignificant for all approaches (less than 1 microsecond, as described later in this section).

Effect of incremental updates on tree efficiency: Figures 8 and 9 show the efficiency of the aggregated tree as incremental updates are applied without an intervening call to *snapshot*. In Figure 8, starting from an optimal efficiency of around 37.5%, the efficiency degrades by less than a percent even after 183,719 updates

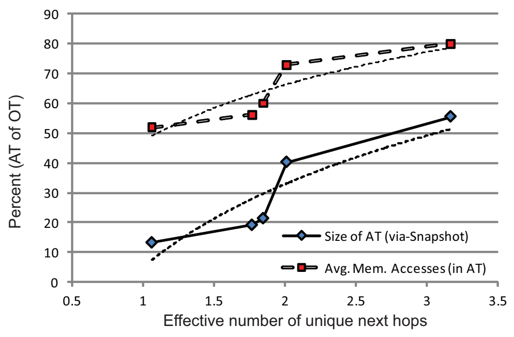

Figure 7: Number of entries and average memory accesses in the AT (as a percent of those in OT) after call to *snapshot*. Results are shown for five Provider ARs (AR-1 through AR-5) with varying number of *effective* nexthops (horizontal axis). The dotted lines indicate the trend.

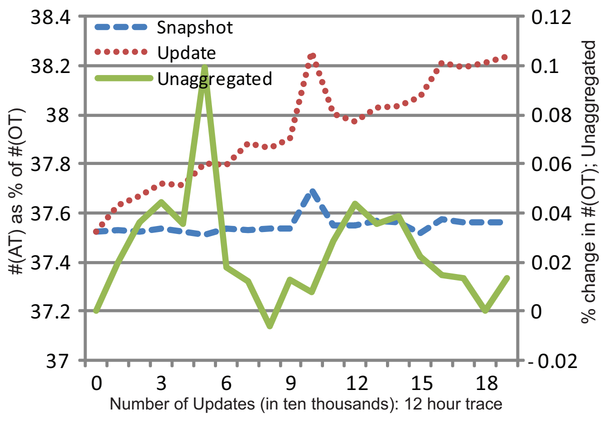

Figure 8: Efficiency of the AT on IGR-1 with increasing number of updates. The optimal size of AT (that would have resulted if *snapshot* was called after every update) is shown for reference. Vertical axis on the right captures the variation in the size of the OT, relative to the starting value.

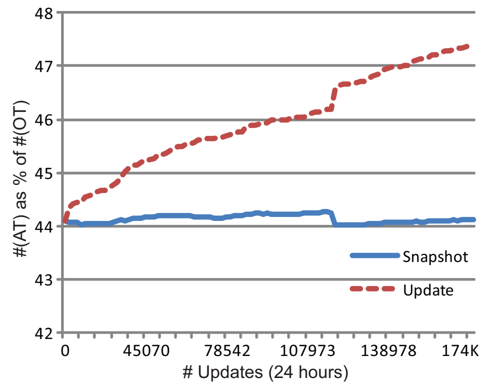

Figure 9: Results (Routeviews, Dec. 2006) showing the progressive variation in AT as the updates are incorporated using our route selection policy. Variation in the size of an optimally-aggregated table is also shown for reference. For other years from 2001 to 2010, the results were similar.

received over a 12 hour period. Likewise figure 9 shows only a few percentage points degradation after tens of thousands of updates gathered over a 24 hour period.

This result so far suggests that a call to *snapshot* can be made infrequently (many hours). Note also that the time it takes to run *snapshot* does not increase with the number of intervening updates, lending additional support to this suggestion. It is possible, however, that running snapshot after a large number of updates produces an unacceptably large number of changes to the FIB. We discuss this next.

Effect of incremental updates on FIB downloads: Figure 10 shows the effect of both incremental updates and snapshots on the number of changes to the FIB, called *FIB downloads*. This figure is for the experiment for IGR-1 with 183719 updates. The x-axis of both graphs gives the number of incremental updates between consecutive calls to *snapshot* during the course of each experiment. The line labeled Update in the upper graph shows the number of FIB downloads due to the incremental updates. This line is essentially horizontal, and shows that for each incremental update, there are about 0.63 ($\sim$120,000/183,719) FIB downloads. The line labeled Snapshot shows the FIB downloads that take place as a result of the snapshot—i.e. the delta between the old and new aggregated trees. From this line, we see that the total number of FIB downloads due to snapshots decreases as the spacing between snapshots increases. However, as the lower graph shows (line labeled Snapshot Burst), the average number of FIB downloads per snapshot actually increases with the number of intervening updates.

The impact of a burst of FIB downloads on the FIB operation depends on the FIB architecture. We note, however, that routers must be capable of responding quickly to a massive number of FIB changes, for instance because a link going down may change the path to a large number of destinations. We measured the time to incorporate an update for the service provider IGRs on a general purpose Core 2 Duo (3GHz, 1333Mhz, 6MB) machine with 4GB of RAM (2 x 800MHz Dual-Channel DDR2 NON-ECC), and discovered that, on average, it took less than one microsecond to incorporate an update. Practically speaking, a router vendor needs to decide how many consecutive FIB downloads are acceptable, and then run the snapshot often enough to stay under this number. The lower graph of Figure 10 shows that even after 20,000 updates, a snapshot produces only 2000 FIB downloads. This corresponds to over one hour between snapshots for the IGR-1 trace.

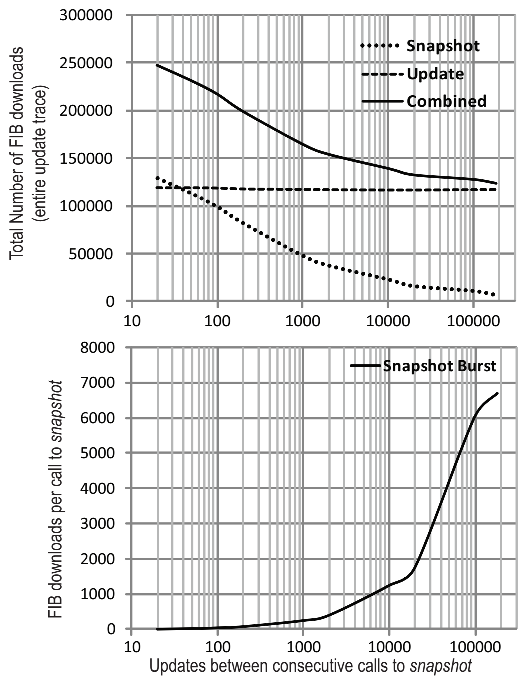

Figure 10: Effect of number of updates between snapshots on FIB downloads (IGR-1). Upper graph shows total number of FIB downloads due to calls made to updates and snapshots respectively over the full run (183719 updates over 12 hours). Lower graph shows the number of FIB downloads per snapshot. In both graphs, the horizontal axis represents the number of updates between two consecutive calls to *snapshot*.

Effect of snapshot on FIB download delay: During a snapshot, received updates to SMALTA are queued up, and only processed after the snapshot completes. This means that every hour or so, convergence is slowed by the time it takes to run the snapshot. [9] reported snapshot run times of roughly 200ms for RouteViews tables (a few 10s of nexthops). We had similar results for a similar number of nexthops. However, snapshot takes longer to run with an increase in the number of nexthops, so we ran snapshot using data from a Tier-1 provider router with a large number of IGP nexthops (roughly 650). In this case, snapshot takes roughly one second to run with a standard desktop computer. In other words, the practical cost of running SMALTA is a sub-second delay in FIB downloads every few hours. Given that iBGP convergence times are often multi-second, this added cost is entirely acceptable.

5. IMPLEMENTATION

We have implemented SMALTA by adding fewer than 2000 lines of code to the Quagga software router code on Debian Linux. Our implementation directly maps to the abstract description in Figure 1 which shows that a software layer is inserted between the RIB and FIB data structures. In our code, this layer has been added inside the *zebra daemon* and takes over all the communication to the kernel. We allow the activation of SMALTA at this layer through the router CLI.

In Quagga, the protocols are implemented as daemons, each of which communicates with zebra which, in turn, is responsible for maintaining the kernel table (i.e., the FIB). To this end, zebra communicates with the kernel via netlink socket by calling the rib_install_kernel() and rib_uninstall_kernel() functions for installing and removing prefixes from the kernel table. By inserting the SMALTA layer, we change the behavior of these functions so that rather than communicating with the kernel directly the said functions re-route the updates to the SMALTA algorithms. At this point, SMALTA is supplied with the updates and the changes are communicated to the kernel via zebra's open netlink socket[3]. In all, the changes are local to the zebra code and do not touch any other protocol code.

We may note that an implementation of SMALTA is dependent on the specific platform architecture used by a vendor. However, based on our experience with Quagga (which is architecturally similar to many proprietary implementations), we expect that deploying SMALTA would require minimal changes to the code base of routers from commercial vendors.

6. RELATED WORK

In 1996, Richardson proposed the idea and a simple implementation of FIB aggregation [13], however its effectiveness on real-world forwarding tables was not determined. In 1999, Draves et al. designed a FIB Aggregation scheme that is provably optimal by the number of entries in the table [4]. Their algorithm, however, requires that an aggregated table be completely recomputed from scratch with each change to the original table. FIB aggregation and related algorithms have also been widely discussed, albeit informally, in the IETF Routing Research Group (RRG) meetings and mailing lists [8]. In 2009–2010, Liu et al. proposed four FIB aggregation algorithms with differing levels of complexity and performance [21, 22]. Two of these (so-called Level-3 and Level-4) have very good compression, but require that non-routable destinations be 'whiteholed' by assigning nexthops to them, potentially causing routing loops [16]. The other two (Level-1 and Level-2) do not aggregate as well as SMALTA. The work closest to SMALTA is also by Liu et al. in 2010 [9]. The algorithms used in this work are not fully specified and may actually lead to routing incorrectness, as detailed in [20].

There is a substantial body of work on fast FIB lookup algorithms for longest-match tables. Examples include [3, 5, 7, 6, 2, 17], but there are many others. These papers strive for extremely fast lookup times while keeping memory and update times small. SMALTA shares these goals. Our view is that SMALTA is complementary to this work and provides savings in memory and lookup times above and beyond what is provided by the FIB lookup algorithms.

Of course, there is an enormous volume of work that attempts to shrink the size of the RIB itself, and by extension, the FIB. This work necessarily requires changes to router operation, either its configuration or more typically its actual protocols. By contrast, SMALTA requires no changes to the external behavior of routers.

Last but not the least, there are FIB suppression methods that distribute FIB entries on various routers, reducing FIB size within each of these routers [1]. These methods require the routers to coordinate and are applicable only for distributed, network-wide deployment. SMALTA does not impose any such requirement and is capable to work on individual routers.

7. CONCLUSION AND FUTURE WORK

We have designed the first proven correct and near-optimal incremental update scheme, SMALTA, based on the optimal "snapshot" scheme, ORTC. Unlike the previous work which evaluated simple FIB aggregation methods for reduction in number of prefix entries, our evaluation of SMALTA focused on (i) the "end-to-end" FIB memory savings, and (ii) the reduction in lookup times, with the popular fast FIB lookup scheme Tree Bitmap. We used both routing tables taken from routers in a Tier-1 provider, as well as 10 years of data from RouteViews. We also explored the relationship between the number of interfaces on a router and the distribution of prefixes over these interfaces.

Our results on storage show that a reasonable "rule

[3]Some kernels may not support the use of netlink socket in which case quagga uses alternate methods to communicate with the kernel. However, use of netlink sockets is the predominant way of communicating with the newer versions of the Linux kernel and is the one used in our implementation.

of thumb" for most non-customer border routers is that Tree Bitmap memory shrinks to about one-half of its original size, while the routing table shrinks to about one-third of its original size. At current DMZ routing table growth rates, this can extend the lifetime of the installed router base by roughly four years. In addition, SMALTA reduces the number of memory accesses for FIB lookup by about 25%, assuming a uniform traffic distribution. Results for typical access routers with the majority of prefixes distributed over very few interfaces (2 or 3) can be substantially better.

Finally, we show that the practical cost of this improvement is that every few hours, there is a sub-second time period during which routing table changes to the FIB are delayed. Our implementation of SMALTA for Quagga shows that it can be cleanly inserted into existing routers with minimal changes to the existing code.

Our numbers for fast FIB memory aggregation improvements are for Tree Bitmap only. Furthermore, our lookup time improvements assume uniform distribution of addresses. Router vendors with other fast FIB lookup algorithms must test SMALTA independently over realistic traffic patterns. While we believe that the results in this paper provide definitive evidence of the value of FIB aggregation, at least for Tree Bitmap, there is still some additional work to do. Intuitively we believe that it should be possible to process updates even while snapshot is running. The idea would be to first insert them "out-of-band" into the FIB while snapshot runs (rather than queue them as we currently do), then process the updates into the aggregated tree, and finally swap the FIB entries for the "out-of-band" entries. Moreover, the impact of changes in BGP to IGP mapping on aggregation in response to path changes in the local AS can be explored further. More broadly, it has been shown that FIB aggregation schemes that allow "whiteholing" of non-routable prefixes can have much better aggregation, but also risk forming routing loops. It would be interesting to consider whether loops could be eliminated in such an approach.

8. REFERENCES

[1] H. Ballani, P. Francis, T. Cao, and J. Wang. Making Routers Last Longer with ViAggre. In *Procedings of USENIX NSDI*, 2009.

[2] A. Basu and G. Narlikar. Fast Incremental Updates for Pipelined Forwarding Engines. In *Proceedings of IEEE Infocom*, 2003.

[3] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small Forwarding Tables for Fast Routing Lookups. In *Procedings of SIGCOMM*, 1997.

[4] R. P. Draves, C. King, S. Venkatachary, and B. D. Zill. Constructing Optimal IP Routing Tables. In *Proceedings of IEEE Infocom*, volume 1, pages 88–97, March 1999.

[5] W. Eatherton, Z. Dittia, and G. Varghese. Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates. *ACM SIGCOMM Compter Communication Review*, 34(2):97–122, 2004.

[6] J. Hasan, S. Cadambi, V. Jakkula, and S. T. Chakradhar. Chisel: A Storage-efficient, Collision-free Hash-based Network Processing Architecture. In *Procedings of ISCA*, 2006.

[7] J. Hasan and T. N. Vijaykumar. Dynamic Pipelining: Making IP-Lookup Truly Scalable. In *Procedings of SIGCOMM*, 2005.

[8] W. Herrin (Original Poster). Opportunistic Topological Aggregation in the RIB–>FIB Calculation? `http://www.ops.ietf.org/lists/rrg/2008/threads.html#01880`.

[9] Y. Liu, X. Zhao, K. Nam, L. Wang, and B. Zhang. Incremental Forwarding Table Aggregation. In *Proceedings of IEEE Globecom*, 2010.

[10] J. Moy, P. Pillay-Esnault, and A. Lindem. Graceful OSPF Restart. `http://www.ietf.org/rfc/rfc3623`, 2003.

[11] Next Generation BGP Monitor. `http://bgpmon.netsec.colostate.edu`.

[12] Quagga. Routing Suite. `http://www.quagga.net/`.

[13] S. J. Richardson. Vertical Aggregation: A Strategy for FIB Reduction (Internet Draft). `http://tools.ietf.org/html/draft-richardson-fib-reduction-00`, 1996.

[14] Routeviews. Project Page. `http://www.routeviews.org/`.

[15] S. R. Sangli, E. Chen, R. Fernando, J. G. Scudder, and Y. Rekhter. Graceful Restart Mechanism for BGP. `http://www.ietf.org/rfc/rfc4724`, 2007.

[16] J. Scudder. Communication on IETF WG (GROW) Mailing List. `http://www.ietf.org/mail-archive/web/grow/current/msg01592.html`, 2009.

[17] S. Sikka and G. Varghese. Memory-Efficient State Lookups with Fast Updates. In *Procedings of SIGCOMM*, 2000.

[18] A. Tariq, S. Jawad, and Z. Uzmi. TaCo: Semantic Equivalence of IP Prefix Tables. In *Proceedings of IEEE ICCCN*, 2011.

[19] Z. Uzmi, A. Tariq, and P. Francis. FIB Aggregation with SMALTA (Internet Draft). `http://tools.ietf.org/html/draft-uzmi-smalta-01`, 2011.

[20] Z. Uzmi et. al. Practical and Near-Optimal FIB Aggregation using SMALTA (Tech Report). `http://www.mpi-sws.org/~zartash/TR-MPI-SMALTA.pdf`, 2011.

[21] B. Zhang, L. Wang, X. Zhao, Y. Liu, and L. Zhang. FIB Aggregation (Internet Draft). `http://tools.ietf.org/html/draft-zhang-fibaggregation-02`, 2009.

[22] X. Zhao, Y. Liu, L. Wang, and B. Zhang. On the Aggregatability of Router Forwarding Tables. In *Proceedings of IEEE Infocom*, 2010.