Surya Detailed Design Specification

Sangeeta Misra
Sangeeta.Misra@sun.com
Sowmini Varadhan
Sowmini.Varadhan@sun.com
Network Performance Team
Network Performance and I/O
Solaris Software

Version 1.2

08 March 2006

Abstract

Surya project aims to improve IPv4 forwarding path scalability. Improving forwarding scalability enables a Solaris machine to forward a higher number of packets per second to a greater number of destinations described in the forwarding table.

The project delivers a faster forwarding table lookup scheme and a streamlined IPv4 forwarding path. These improvements, when combined with soft-ring(PSARC 2005/654) and Crossbow's polling implementation, will vastly enhance Solaris forwarding throughput performance. Crossbow's polling-based feature will aim to solve receive livelock problems that are common in interrupt-driven kernels, like Solaris (for more details please see reference 5) In addition, Surya will also deliver APIs for IP Filter that will allow simplification of IP Filter implementation.

The changes addressed by this project improves IPv4 forwarding only and thus does not address IPv6 forwarding performance.

Sun Microsystems, Incorporated.

Contents

1 Introduction

This document records the design of Surya project. Section 2 discusses the goals of the project. Section 3 identifies the non-goals of this project. Section 4 discusses the details of the new Forwarding Information Base scheme (FIB). Section 5 provides an overview of current Solaris IPv4 forwarding path. Section 6 covers the detailed design of the changes necessary to improve the IPv4 forwarding path. The reader will find it useful to consult Appendix A when reading section 6. Section 7 describes new IP filter API. Section 8 discusses future directions we may want to pursue. Section 9 provides references consulted when producing this design. Section 10 identifies people who provided much appreciated help during this project. Two appendices are provided. Appendix A describes the new code flow for the IPv4 forwarding path. Appendix B describes the analysis we did of two FIB schemes to aid us in choosing the best scheme for Solaris.

2 Goals

The goals of this project are:

1. Improving IPv4 forwarding scalability - this entails the following:
   - A faster FIB scheme.
   - An optimized IPv4 forwarding path.
   - Optimization of timer-based ICMP redirect processing in the FIB.
2. Delivering APIs for IP Filter. This project will deliver two interfaces to be used by IP Filter.
3. Restructuring the link layer address storage in ire_t data structure and insertion of incomplete ires (ie ires whose link layer address is not fully-resolved) in IRE cache table via the forwarding path. The latter scheme should be, in the future, extended to the host path to greatly simplify IPsec and other implementations in IP module. Please refer to section 6 for this discussion.

3 Non-goals

The following items are non-goals for this project:

1. Improving IPv6 forwarding performance. We plan to address this in future project by extending the implementations of this project. Surya will not impact current IPv6 functionality, and IP will continue to support IPv6 as it currently does.

2. Merging of the ARP kernel module functionality into the IP module. Since Surya's focus is to improve IPv4 forwarding performance, and ARP-IP merge would not have any impact on that goal, the latter is out of the scope of this project.

4 New Forwarding Information Base algorithm

4.1 Existing FIB scheme

Currently in Solaris, the IP forwarding lookup is based on a per-netmask hash table. For a 32-bit IP address, there are 33 netmasks possible and one hash bucket dedicated per netmask. As an example case, in current Solaris, if the following route add commands are executed on a system:

```
# route add 10.10.10.50/32 172.16.0.21
# route add 10.10.10.50/32 172.16.1.21
```

the corresponding FIB layout would look like this:

```
   ip_forwarding_table
   .-----------------.
0  |                 |
   -------------------
   |      :          |
   -------------------
32 |      o----------+--.
    -----------------   |
                        V
          array of irb_t structures
         -------------------
       0 |                 |
         |-----------------|
         |      :          |
         |-----------------|
    [#B] | irb_ire  -------+--->           ire_t
         |  :              |<-. ------------------------------------
         |  :              |  | | ire_next   ---------------------+------.
         | irb_ire_cnt = 2 |  | | IRE_HOST                        |      |
         |  :              |  | | ire_addr = 10.10.10.50          |      |
         |-----------------|  | | ire_mask = 255.255.255.255      |      |
         |      :          |  | | ire_gateway_addr = 172.16.0.21  |      |
         |      :          |  ‘-+ ire_bucket                      |      |
         |-----------------|  | |    :                            |      |
     N   |                 |  | ------------------------------------     |
         -------------------  |                                          |
 Note :                       |                                          |
 N = IP_FTABLE_HASH_SIZE - 1; |                                          |
 [#B] in the figure above  is |                                          |
 computed as                  |                                          |
 IRE_ADDR_HASH(ire_addr, ip_ftable_hash_size)                            |
                              |                                          |
                              |                                          |
                              |          ire_t          <----------------’
                              |  ------------------------------------
                              |  | ire_next  ----------------------+-----.
                              |  | IRE_HOST                        |     |
                              |  | ire_addr = 10.10.10.50          |     =
                              |  | ire_mask = 255.255.255.255      |
                              |  | ire_gateway_addr = 172.16.1.21  |
                              ‘--+ ire_bucket                      |
                                 |     :                           |
                                 ------------------------------------
```

In order to find the outgoing interface for a packet, the longest matching prefix from the FIB must be looked up. In the current scheme, the search begins at the bottom of the hash table i.e., corresponding to the /32 netmask (i.e., 255.255.255.255). A value is computed to identify the appropriate bucket, and then the linked list of entries is walked to find a matching entry. This process is repeated for each netmask bucket in descending order (32, 31, 30 ..) till a match is found.

This algorithm does not scale well for large routing tables, when each netmask bucket containing potentially long lists of route entries has to be traversed.

4.2 Prototyping of alternative schemes

We looked into various lookup algorithms in search of a scheme that would be appropriate for general-purpose systems. We found most schemes to be suited for embedded systems, hardware and distributed implementations. We thus decided to look at PATRICIA trees(aka *Radix* tree) and a variant of the multi-bit trie. The algorithms tested were:

a. Address-directed FIB scheme that has a multi-bit trie structure, in which segments of the IP address index in, to lead to the last level of the trie which points to the list of ires (possibly duplicate). A search for the most specific route would search the trie starting with the destination address. The backtracking is done by masking the address with zeros progressively, from more specific prefixes to less specific prefixes and retrying the search in the trie. An 8-bit trie is used. Thus there are 256 entries per node, and a total of 4 levels. At each node, the backtracking is speeded up by having a bitmask for the entries to find out whether it is non-zero. This scheme is not a software re-implementation of a pure multi-bit trie as described in reference 4 as most of the optimizations listed in that document have patent rights attached to them, and are geared towards hardware implementation of multibit-trie.

b. FreeBSD's implementation of Radix tree. Note that the Radix tree implementations in FreeBSD, OpenBSD and NetBSD are essentially the same (with minor differences). Thus FreeBSD was arbitrarily chosen. For a detailed description of Radix tree, please refer to reference 2 and 3.

A kernel prototype implementation of each scheme was implemented. Each algorithm was installed on a SunFire V40Z quad 2.4 GHz cpu AMD Opteron System and tested against the Spirent Smartbits Tera Routing Tester. The results containing the throughput and memory usage data is listed in Appendix B.

Due to following considerations:

a. No significant difference in throughput and memory usage was noted between the Address-directed FIB scheme and Radix tree implementation.

b. The FreeBSD implementation is well documented, understood, and has had exposure for years.

c. The FreeBSD implementation can be easily extended to IPv6 forwarding scheme and implementation of ECMP(Equal-Cost Multipath Protocol) in future projects.

the FreeBSD scheme was chosen for the FIB algorithm.

4.3 Solaris glue points to FreeBSD's radix tree structure

A description of the data structures used to represent the Radix tree in BSD can be found in reference 3.

As an example case, Solaris implementation of radix tree (with glued ire_t data structure) on a system where the following route add commands are executed:

```
# route add 10.10.10.50/32 172.16.0.21
# route add 10.10.10.50/32 172.16.1.21
```

would look like this:

```
               radix_node_head
                --------------
                | radix_node |
                | (left end) |
                -------------
          .---- |            | ----.
          |     -------------      |
          |     | radix_node |     |
          |     | (right end)|     |
          |     --------------     |
          V                        V
        (left)                    (right)
          :                        :
          :                        :
   .->  rt_entry
   |  ------------------------
   |  | radix_node {}        |
   |  | rn_Key = 10.10.10.50 |
   |  ------------------------
   |  | radix_node {}        |
   |  | rn_bit = 46          |
   |  ------------------------
   |  | rt_dst = 10.10.10.50 |
   |  ------------------------                    ire_t
.--(->|     irb_ire   -------+-+---->  .--------------------------------.
|  |  |      :               | |       |       ire_next             ----+--.
|  |  |      :               | |       |       IRE_HOST                 |  |
|  `--+---- irb_rt           | |       | ire_addr = 10.10.10.50         |  |
|     |     irb_rr_origin  --+-'       | ire_mask = 255.255.255.255     |  |
|     ------------------------         | ire_gateway_addr = 172.16.0.21 |  |
+--------------------------------------+---   ire_bucket                |  |
|                                      |          :                     |  |
|                                      `--------------------------------'  |
|                                                                          |
|                                                 ire_t  <-----------------'
|                                       .--------------------------------.
|                                      |       ire_next             ----+--.
|                                      |       IRE_HOST                 |  |
|                                      | ire_addr = 10.10.10.50         |  |
|                                      | ire_mask = 255.255.255.255     |  =
|                                      | ire_gateway_addr = 172.16.1.21 |
`--------------------------------------+---   ire_bucket                |
                                       |          :                     |
                                       `--------------------------------'
```

NOTE: For a detailed description of data structure radix_node_head and radix_node, please refer to reference 3.

The routing table has a radix_node_head and all the nodes in the routing tree, both the internal nodes and the leaves, are radix_node structures. The routing table tree is built from rt_entry structures. Each rt_entry structure contains two radix_node structures which attach it to the radix tree: one radix_node structure is an internal node, corresponding to the bit to be tested, and the leaf node itself containing information about the internet route.

In order to glue Solaris ire_t data structure, we have modified the rt_entry structure (note that it's different from BSD's rtentry structure) The layout of the rt_entry structure is shown below:

```
struct rt_entry {
        struct  radix_node rt_nodes[2]; /*tree glue */

        /*
         *struct rt_entry must begin with a struct
         * radix_node (or two!) to a 'struct rt_entry
         */
        struct rt_sockaddr rt_dst;

        /*
         * multiple routes to same dest/mask via
         * varying gate/ifp are stored in the
         * rt_irb bucket.
         */
        irb_t rt_irb;
};
```

Detailed information about the route is stored in a linked list of ire_t structures which may be accessed by following the `rt_irb->irb_ire` pointer. Thus, multiple routes to the same IPv4 destination and netmask are stored in the same rt_irb. Further details of the contents of the rt_irb and modifications to the existing ire_t structures are discussed in next section.

The functions in the BSD implementation that modify or search the radix tree expect to be passed a pointer (void *)varg to the search key such that the length of the key may be obtained in the first byte pointed to by varg. Since ire_t structures store IP addresses as ipaddr_t structures, in order to efficiently call the radix functions, the search key is stored in the rt_entry in the rt_dst as a rt_sockaddr structure that is defined as:

```
struct rt_sockaddr  {
        uint8_t         rt_sin_len;
        uint8_t         rt_sin_family;
        uint16_t        rt_sin_port;
```

```
                    struct in_addr  rt_sin_addr;
                    char            rt_sin_zero[8];
        };
```

4.4 Introducing rt_entry and modifications to irb_t

Each leaf of the radix tree (where the radix_node has rn_b < 0) is a rt_entry structure containing a bucket of Internet routing entries. The existing definition of ire_t has been re-used without change for the routing entries themselves. Although the irb_t type continues to be used as the bucket data structure, it has been modified to accommodate the requirements introduced by the radix tree. These modifications are listed below.

- Maintain a pointer irb_rt to the rt_entry containing the bucket.
- irb_t structures were never deleted or freed in the pre-Surya implementations of the FIB, but are now dynamically allocated and freed in Surya. Since an irb_t may only be freed when there is no reference to the for the irb_ire list, a new irb_marks flag of IRB_MARK_DEAD has been introduced. This marker is set on the rt_irb when a route is deleted from the the radix tree, indicating that the bucket is no longer attached to the tree, and allows IRB_REFRELE to safely free the associated rt_entry when all the ire's in the bucket have been removed.

4.5 Default route handling in new FIB scheme

Pre-Surya implementation of the FIB provides for simplistic load balancing scheme that round-robins through the list of default routers. The list of default routers is accessed by looking at the contents of ip_forwarding_table[0], and the global index ip_ire_default_index indicates the next list member at which the round-robin search should start. Further, if TCP notices problems that causes excessive retransmits (i.e., a problematic router) it invokes ip_ire_delete() via tcp_ip_notify() to adjust the round-robin search so that the problematic router is skipped.

The above scheme has the limitation that it does not extend well to allow the round-robining of general (i.e., nondefault) prefix routes. Further, in the radix tree implementation, the ip_forwarding_table[0] pointer to default routes is no longer supported.

Instead, a new field, irb_rr_origin is maintained in every irb_t structure, to track the next ire at which the round-robin search should start. When problematic routers are detected in ip_ire_delete(), the irb_rr_origin is updated appropriately. Round-robin itself is implemented in the new function ire_round_robin().

4.6 Locks and synchronization structures

BSD implementations of the PATRICIA tree provide support for a mutex that protects the radix_node_head. Surya refines this mutex to be a rw_lock, with the lock being held as RW_WRITER during route addition or deletion, and held in RW_READER mode elsewhere. The rationale for this locking scheme was to optimize access for multiple readers, which was expected to be the more commonly encountered case. Most systems will likely have just a single thread (the user-space routing daemon) that is actually writing to the kernel's FIB. A reader-writer lock implements an implicit mutex, and, if lookup is a big fraction of the time involved in forwarding and allows multiple cpus to handle incoming packets.

4.7 Route addition

When adding an entry to the FIB in ire_add_v4(), the function ire_get_bucket() is called to obtain the rt_irb for the route to be added. ire_get_bucket() first attempts to add a node to the radix tree by invoking rn_addroute(). If the route already exists, rn_addroute() returns NULL, in which case ire_get_bucket() calls rn_match() to return the existing route. The value of rt_irb is returned to the caller.

4.8 Route deletion

The existing function, ire_delete() is invoked for both cache-table and forwarding-table ire entries. In the latter case, when the ire is deleted, if there are no other ire entries in the bucket, the bucket itself, and the rt_entry it belongs to, must be removed from the radix tree. Cache table ire entries must continue to delete the entry following pre-Surya procedures which are now contained in the function ire_delete1().

Thus the function ire_delete() now executes as follows(note that the following code fragment is not the complete illustration of this function):

```
/*
 * Delete the specified IRE.
 */
void
ire_delete(ire_t *ire)
{
        struct radix_node *rn = NULL;
        struct rt_sockaddr rdst, rmask;
        struct rt_entry *rt;

        if ((ire->ire_type & IRE_FORWARDTABLE) == 0) {
                ire_delete1(ire);
                return;
```

```
        }

        /* first remove it from the radix tree. */
        ....

        if (ire->ire_bucket != NULL &&
            ire->ire_bucket->irb_ire_cnt == 1) {
                /*
                 * only one ire in this bucket;
                 * can remove irb from tree
                 */
                rn = ipftable->rnh_deladdr(&rdst, &rmask,
                    ipftable);
        }
        .....

                /* got a free standing irb; mark it dead */
                rt = (struct rt_entry *)rn;
                rt->rt_irb.irb_marks |= IRB_MARK_DEAD;
                ip1dbg(("mark rt 0x%lx dead\n", (ulong_t)rt));
        }
        ire_delete1(ire);
}
```

The marker IRB_MARK_DEAD is described in sub-section 4.4, and is set when a leaf node is removed from the tree, allowing IRB_REFRELE to delete the node.

4.9 Route lookup

Forwarding table lookup is done by invoking an enhanced version of rn_match() function. The modifications made to rn_match() are described below.

By default, the BSD code for tree-search returns the longest matching prefix to the caller. In Solaris, ire_ftable_lookup can be called with complex permutations of IRE_MATCH_* flags so that the matched key may not necessarily be the longest matching prefix. In order to support these queries, the BSD radix code for tree search was modified to allow callers to pass in a (possibly null) function pointer matchf() to the rn_match() function, so that matchf() is invoked on every matching leaf. If the the supplied function pointer matchf() is NULL, then the default BSD search(ie. longest matching prefix) is performed, otherwise search will include lookup for matching prefixes.

Modifications to the round-robin search for default routers has been discussed earlier in sub-section 4.5.

4.10 Routing table walk

The radix tree may be walked by invoking the function rn_walktree() which already accounts for the case when a node is deleted by the traversing function, while the tree is being walked. However, since there is a possibility of lock recursion if the function invoked by rn_walktree() attempts to lock the radix tree, the radix_node_head read lock is released before invoking function pointers from rn_walktree().

5 Overview of existing IPv4 forwarding code path

In current Solaris, processing of forwarded packets proceeds as follows:

1. Search cache table to find a ire cache entry for the ipha_dst
2. If no cache entry is found in step 1, look up the FIB to find the appropriate route for ipha_dst. Assume that this search produces a route through some gateway *G*.
3. Try to find an ire cache entry for *G*. If none exists, send a request to the external resolver. This request contains a chain of mblks containing:

   ```
   ---------------       ---------------     -----------
   |             |       |             |     |         |
   | dl_unitdata |  -->  |  mblk w/ ire| --> |   pkt   |
   | request     |       |  "ire_mp"   |     |         |
   ---------------       ---------------     -----------
   ```

   Where the ire *ire_mp* contains a template for the ire cache entry for *G*.
4. When the external resolver provides the information to complete ire_mp, the ire cache entry for *G* is added to the cache table.
5. The code now attempts to add an ire cache entry for the off-link dst, ipha_dst, itself. After completion of this step, the packet's ip header is processed (ttl adjustment, qos etc.) and the packet is sent out.

The above scheme has the following limitations:

a. The creation of per-dst cache entry for the off-link destination for every forwarded packet in Step 5 results in additional looping through IP outbound code path.

b. In the above scheme, ires of the next hop are inserted into the cache table only after its link layer address is fully resolved. There is no need to have this limitation. In fact in future when IPsec implements policy hooks in the forwarding path, the removal of this limitation could simplify IPsec's implementation.

In the rearchitected IPv4 forwarding path addresses the above limitations thus:

1. By improving the lookup speed of the FIB, we have eliminated the need to keep per-dst cache entry for the off-link destination for every forwarded packet.
2. We insert ires of next hops without link layer information in the IRE cache table (we call these ires in the cache table *incomplete ires*) and proceed with the rest of packet processing. Eventually Surya's new packet transmit routine, ip_xmit_v4() attempts to send a packet to the driver. If it now finds that there is no link layer information, it triggers the ARP resolution process, and queues the packet in an internal queue and then sends queued packets out once the ARP resolution is over.

6 IPv4 forwarding Path Changes

6.1 Addition of nce_ts for IPv4 IRES

6.1.1 Motivation for using nce_ts for IPv4 ires

To implement this new architecture, and in an attempt to unify the code path with that chosen for IPv6 Neighbor-discovery, we will utilize the existing nce_t data structure and the member `ire->ire_nce` for IPv4 ires.

Note that only specific members of the nce_t data struct are relevant for IPv4 ires:

a. `ire->ire_nce->nce_state` will track the link layer resolution status. The IPv4 related status values are thus:

   ND_INITIAL indicates that the sending of DL_UNITDATA request for the link layer address resolution is pending.

   ND_INCOMPLETE indicates that a DL_UNITDATA request has been sent to the resolver, and link layer address resolution is pending.

   ND_REACHABLE indicates that the link layer resolution is complete, and the layer 2 address is available.

   NOTE: ires of type IRE_PREFIX and IRE_DEFAULT remain in ND_INITIAL state permanently.

b. the `ire->ire_nce->nce_qd_mp` will be used as the internal queue to queue data packets for the ire while waiting for its ARP resolution to complete.

c. `ire->ire_nce->nce_res_mp` and `ire->ire_nce->nce_fp_mp` will be used for DL_UNITDATA request and responses, making ire_dlureq_mp/fp_mp unused fields. In a future release we plan to remove ire_dlureq_mp/fp_mp from the ire_t data structure.

Note that in Surya, the IPv4 nces will not be used as a link layer address cache (ie: ace_t will continue to be used for that).

6.2 Initialization of ire_nce for IPv4 ire's

The ire_nce field in the ire_t will track the information necessary for resolving the link layer corresponding to the outgoing interface that is tracked by ire_stq. When all the layer-2 information necessary to send a packet is available, the ire will be termed as *complete*, and the ire_nce will be defined to be reachable. As mentioned above, the state of the ire_nce is determined from the nce_state field, and will be set to ND_INITIAL, ND_INCOMPLETE or ND_REACHABLE.

The ire_nce is initialized by calling the function ire_nce_init() from ire_init_common(). The contents of this field for each ire_type are described as follows:

- ire_t entries with NULL values of the send queue, ire_stq (e.g., IRE_LOCAL, IRE_LOOPBACK) have a NULL ire_nce. The resolver information in these cases is deterministic. ire_t entries with null ire_nce fields are assumed to be ND_REACHABLE by definition.
- Non-loopback IRE_BROADCAST ire's have the nce_res_mp set to the pre-computed template generated in ip_ll_subnet_defaults(). The ire_nce has nce_addr/nce_mask set to the IPv4-mapped-IPv6 addr corresponding to ire_addr/ire_mask. Since no DL_UNITDATA request needs to be sent to the resolver for these ire entries, their state is initialized to ND_REACHABLE in ire_init_common. Note that a fast-path probe is sent out to the network drivers to obtain the fast-path header for these ire's.
- ire entries for subnet prefixes (IRE_DEFAULT, IRE_PREFIX) have the ire_nce initialized with the nce_addr and nce_mask set to the IPv4-mapped-IPv6 addr corresponding to the ire_addr and ire_mask. The nce_res_mp for these ire types is used to track a copy of the template DL_UNITDATA message. If a template for the resolver_mp (mblk to be used as DL_UNITDATA message) is not passed in, ire_nce_init() will set the nce_res_mp to a copy of the ill_resolver_mp for the outgoing interface. If the interface is of type IRE_IF_RESOLVER, the nce_t is set to ND_INITIAL when the nce_t is created. If the interface is of type IRE_IF_NORESOLVER, the nce_state is set to be ND_REACHABLE.
- IRE_CACHE type ire entries have the ire_nce initialized in ire_nce_init as follows:
  - The nce_addr is set to the IPv4-mapped-IPv6 addr corresponding to the ire_gateway_addr, for indirect routes and to the IPv4-mapped-IPv6 addr corresponding to the ire_addr for on-link destinations.

  - The nce_mask is set to ipv6_all_ones.
  - If the outgoing interface for the ire is of type IRE_IF_RESOLVER (i.e. external resolver has to be called to resolve the layer 2 header), the nce_t is set to ND_INITIAL when the nce_t is created; if the outgoing interface for the ire is of type IRE_IF_NORESOLVER (e.g., tunnel or point to point interfaces), the nce_t is set to ND_REACHABLE when the nce_t is created. As with IRE_DEFAULT/IRE_PREFIX ire's, the nce_res_mp is set to the template res_mp if one is passed in, or by making a copy of the ill_resolver_mp for the outgoing interface when no res_mp is passed in.

    After a DL_UNITDATA request is dispatched to the external resolver for IRE_CACHE entries, the nce_state transitions to ND_INCOMPLETE. When layer 2 resolution is completed, and a DL_UNITDATA response is received from the resolver, the nce_res_mp is set to the mblk with the DL_UNITDATA response, and the nce_state transitions to ND_REACHABLE, at which point the ire is defined as *complete*.

NOTE: The ire to nce mapping is many-one; e.g., if we have a subnet route that is added by the command:

```
# route add 10.10.10.0/24 129.23.45.1
```

then the IRE_CACHE entries for 10.10.10.1, 10.10.10.2 (created as a result of local traffic sent to 10.10.10.1 or 10.10.10.2) and 129.23.45.1 will all hold a pointer to an nce_t with nce_addr containing the IPv4-mapped-IPv6 addr for 129.23.45.1.

6.3 Modifications to IPv6 Neighbor Discovery code

Neighbor cache entries added by Surya from the IPv4 packet processing path are managed in a hash table that is kept separate from the IPv6 hash table. Thus ip_ndp.c now defines two entries:

```
static nce_t *nce_hash_tbl_v6[];
static nce_t *nce_hash_tbl_v4[];
```

with IRE_ADDR_HASH() defined as the hash function used to access buckets in nce_hash_tbl_v4[]. As a result of the addition of separate hash tables, nce_lookup_addr() has been modified so that the caller passes in the nce_t after computing the appropriate hash-bucket.

Neighbor cache entries for IPv4 are managed by the following functions (which have _v6 counterparts):

- Neighbor cache entries for IPv4 are added by ndp_add_v4(). This function takes inaddr_t arguments for the address and mask. The address is stored in the nce_t as a IPv4-mapped-IPv6 address. If the mask is IP_HOST_MASK, then the nce_mask is set to ipv6_all_ones; otherwise it is set to the IPv4-mapped-IPv6 address corresponding to the mask passed in.
- ndp_lookup_then_add_v4() which differs from the _v6 counterpart by allowing the caller to pass in precomputed values for nce_res_mp and nce_fp_mp when adding nce_t.
- The nce_hash_tbl_v4[] nce_t's may be walked by invoking the function ndp_walk_impl_v4(). Analogous to ndp_g_walker, a global flag, v4ll_g_walker has been added to prevent ndp_delete() from unlinking and freeing nce's while the list is being walked. Modifications to v4ll_g_walker are protected by v4ll_g_lock. In addition, a boolean v4llg_walker_cleanup has been added as the IPv4 analog of ndp_g_walker_cleanup.
- nce_queue_mp has been broken up into a v6-specific function, and nce_queue_mp_common which is shared with ipv4.

The formulating of IPv4 ires with `ire->ire_nce` causes a IPv4 ire to hold reference to the nce_t that has been installed in a nce_hash_tbl_v4[] entry.

When a link layer address resolution request is sent to the external resolver, the ARP query message chain represents an ire that *may* hold a reference to an nce_t structure, if the ire was added as an incomplete ire to the cache table. If the link layer address resolution fails, this reference will need to be released as part of failure recovery and cleanup. Refer to the next Section's discussion on the free routine, ire_freemblk() to see how this is achieved.

6.4 Rearchitected IPv4 forwarding code path

In the rearchitected IPv4 forwarding path, processing of forwarded packets will proceed as follows:

1. Search cache table to find a ire cache entry for the ipha_dst
2. If no cache entry is found in step 1, look up the FIB to find the appropriate route for ipha_dst. Assume that this search produces a route through some gateway *G*.
3. Try to find an ire cache entry for *G*. If none exists, create an ire cache entry for the gateway. Mark this ire entry as incomplete, indicating that its link-layer address is yet to be resolved, and insert it into the IRE cache table.

4. Complete all of the packet processing that does not require the link-layer address, and queue the packet into the incomplete ire's internal queue.

5. Send a request to the external resolver to have the ire's link-layer address resolved.

6. Once the external resolver provides the link-layer address, the ire is marked as complete and all packets that were queued in its internal queue are sent out.

The code flow of the rearchitected IPv4 forwarding path is depicted in Appendix A. The diagram illustrates both the fast and slow (FWD_FASTPATH and FWD_SLOWPATH) paths of the forwarding code path. Please refer to junction points in the diagram to follow the discussion in this section.

The key points of the new IPv4 forwarding code path are the following:

- Unlike current Solaris, cache entries for off-link destinations are no longer inserted into the IRE cache table via the IPv4 forwarding path. Instead, only cache entries of next hops are inserted.

- Unlike the host path, the IPv4 forwarding path will insert incomplete ires into the cache table (refer to JUNCTION A). In fact, the ARP resolution process is delayed until the very end at JUNCTION C in the following code path:

  ```
  ip_xmit_v4()->ire_arpresolve()
  ```

- Note that the ARP message chain (refer to JUNCTION D):

  ```
  ARP_REQ_MBLK-->IRE_MBLK
  ```

  formulated in ire_arpresolve() of the IPv4 forwarding code path, does not include the data packet (unlike in the case of ip_newroute(), ip_newroute_ipif() of the host path). Instead, for the forwarding path, the data packet is already queued in `ire->ire_nce->nce_qd_mp` in nce_queue_mp_common() before the sending of the ARP query.

  Since the incomplete ire has already been added to the cache table, a dummy (*fake*) ire is sent to ARP as part of the ARP message chain. The *fake* ire contains the minimum information required to retrieve the corresponding incomplete ire from the IRE cache table by doing an ire_ctable_lookup(); this is needed to either fill in the link layer address info into the `ire->ire_nce->nce_res_mp` in case of a successful layer 2 address resolution or for cleanup purposes in case of a failed Layer 2 resolution.

The second mblk of the ARP query message chain (this applies to host path as in the case of ip_newroute(), as well as forwarding path, as in the case of ire_arpresolve()), containing the ire information (ie the one labeled as *IRE_MBLK* in picture above) is allocated via esballoc(), with the free() routine set to *ire_freemblk().*

In the case of a failed link layer address resolution, ire_freemblk() can be invoked by ARP (in case of a timeout) or IP. On such an event, ire_freemblk() performs the following cleanup tasks:

a. Retrieval of the incomplete ire in the cache table that corresponds to the *fake* ire in the original message.
b. Sending of icmp unreachables for any queued data packets on the ire's internal queue.
c. Release of resources held by the ire (including the reference on the nce_t, if the ire was added to the cache table).
d. Cleaning up the incomplete ire (and its ire_nce) entry in the IRE cache table.

In the case of a successful link layer resolution, ire_freemblk() can be called by IP in code flow:

```
<ARP>->ip_wput()->ip_output()->ip_wput_nondata()
```

which will free the mblk after processing the ARP response.

- The new function, ip_xmit_v4() is in charge of triggering the ARP querying and queuing of the data packets in ire's internal queue until the ire's link layer address resolution is complete. Once ARP resolution is complete, ip_xmit_v4() is revisited for the second time via code path(refer to JUNCTION E):

  ```
  <ARP>->ip_wput()->ip_output()->ip_wput_nondata()->
      ip_xmit_v4()
  ```

  This time, since the ire's nce_state has changed to ND_REACHABLE, ip_xmit_v4 () processes each queued packet by attaching the link layer header and sending it out on the wire. Note that ip_xmit_v4() does not handle fragmentation, and that task is still handled by ip_wput_frag().

- In Surya, we have introduced a new mblk_t.dblk_t.db_type called IRE_ARPRESOLVE_TYPE, that is distinct from IRE_DB_TYPE. IRE_ARPRESOLVE_TYPE is used by ire_arpresolve() to send ARP query message in the forwarding path for incomplete ires that have already been inserted in the cache table. IRE_DB_TYPE continues to be used by ip_newroute and ip_newroute_ipif in the host path that do not add incomplete ires in the cache table. Thus the handling of the ARP response for each case in ip_wput_nondata() is different:

  - In the case of IRE_DB_TYPE, ip_wput_nondata() calls ire_add_then_send() to insert the completely resolved ire into the cache table and then send the packet that was attached to the ARP request message chain. Surya will preserve this implementation in the host path as is in current Solaris.
  - In the case of IRE_ARPRESOLVE_TYPE, the ire is already present in the cache table. So ip_wput_nondata() calls ip_xmit_v4() which simply processes each queued packet of the ire's `ire_nce->nce_qd_mp` by attaching a link layer header and sending it out on the wire.

6.5 Handling of incomplete ires in the host path

6.5.1 Motivation for insertion of incomplete ires in cache table

As discussed in Section 5, in the existing Solaris model only complete ires are inserted into the cache table in both host and forwarding path. This limitation adds unnecessary complexity to IPsec and other packet processing in the host path. Let us take the case of IPsec.

One of the problems of IPsec policy enforcement is that it has to survive asynchrony and recover state used to protect the packet on the outbound side. One source of asynchrony is the limitation in outbound host path where ip_newroute only inserts complete ires in the cache table. IPsec policy decisions on outbound packets are made after determining the IRE cache entry, because only then are the source and destination addresses known. Since the cache entry insertion requires link layer resolution, IPsec processing is unnecessarily delayed even though it does not require any link layer info. Thus insertion of *incomplete* ire cache entries (that has the source address fixed) by ip_newroute() will allow IPsec processing to occur in parallel with ARP resolution completion.

The long-term goal is to implement incomplete ires in IPv4 and IPv6 host and forwarding paths in IP. Ideally it would be good to implement this full-blown incomplete ire scheme within a single project. However due to code complexity (ie CGTP, IPsec) in host path, this work has to be done in phases. Surya has implemented incomplete ires in the IPv4 forwarding path in a way such that the infrastructure can be extended into the host path in a future project.

6.5.2 Changes to the IP outbound legacy path to handle incomplete ires

Surya has changed the following key functions in IP outbound path:

1. ip_wput_ire()
2. ip_rput_forward()

3. ip_mrtun_forward()

so that at the end of packet processing, instead of calling ip_wput_attach_llhdr(), they each call ip_xmit_v4() and supply the ire and the data packet to it to have the packet sent out on the wire. The function ip_xmit_v4() is designed to handle link layer address resolution, queueing of the packets while address resolution is pending and eventual sending out of the packet.

In the case of ip_wput_ipsec_out(), it will drop a packet if the corresponding ire is incomplete. If, on the other hand, the ire is complete, ip_wput_ipsec_out() will hand over the packet and the ire to ip_xmit_v4() to handle IPsec hardware acceleration. It's too complex to get the IPsec hardware acceleration approach to fit well with ip_xmit_v4() doing ARP without doing IPsec simplification, which is a separate project in itself.

Similarly ip_wput_frag() has been modified, so that early in the function (before fragmentation effort begins), the code checks to see if the supplied ire is incomplete. If it is, and the ire's nce state indicates ND_INITIAL (e.g. ARP request has not been sent out), it calls ip_xmit_v4() to trigger the sending of ARP request for that ire, and drops the packet (and all subsequent packets for that ire, until its link layer address is resolved). Post-ARP resolution, after ire's nce_state changes to ND_REACHABLE, all subsquent large packets for this ire will be fragmented and sent out by ip_wput_frag().

Note that in the case of ip_wput_ipsec_out() and ip_wput_frag() there is a slight risk here, in that, if we have the forwarding path create an incomplete ire, then until the ire is completed, any transmitted IPsec packets, or fragmentable packets will be dropped instead of being queued, waiting for resolution. But the likelihood of a forwarding packet and a wput packet sending to the same destination at the same time and there is not yet be an ARP entry for it is small. Furthermore, if this actually happens, it would be likely that wput would generate multiple packets (and forwarding would also have a train of packets) for that destination. If this is the case, some of them would have been dropped in existing Solaris as well, since ARP only queues a few packets while waiting for resolution

6.5.3 Impact of incomplete ires for callers of ire lookup functions

Note that the only thing missing in an incomplete ire is the link layer address information in the dl_unitdata and fast-path headers; everything else is already initialized at the time of their insertion into the IRE cache table. So with the exception of a few, most consumers of the functions:

`ire_ctable_lookup(), ire_cache_lookup(), ire_route_lookup()`

will be unaffected if the lookup function returns an incomplete ire. We will now discuss the few consumers that are affected, and how they are dealt with:

tcp_send_data() this function already checks ire's fastpath header for non-null value before sending a packet on the wire.

tcp_multisend() Surya has modified this function to check for incomplete ire and if so, send the packet on the IP outbound legacy path.

udp_send_data() Surya has modified this function to check for incomplete ire and if so, send the packet on the IP outbound legacy path.

ip_wput_frag_mdt() In the case of an incomplete ire, Surya's modifications to ip_wput_frag() will disallow calling of this function and the packet will dropped early in in the caller of this function.

ip_sioctl_iocack() Surya has modified this function so that if the call to ire_ctable_lookup() returns an incomplete ire, it's treated as if the lookup had returned a NULL ire.

fr_fastroute() This IP Filter function already checks the ire to see if it has a complete link layer address before attempting to send a packet out into the wire.

pfil_sendbuf() This IP Filter function already checks the ire to see if it has a complete link layer address before attempting to send a packet out into the wire.

Future consumers of the above ire* lookup functions who retrieve the ire to either refer to or use the link layer address from the ire's data structure, must make the following check:

```
if (ire != NULL && ire->ire_nce &&
    ire->ire_nce_nce_state == ND_REACHABLE)
```

before proceeding. If the caller of the ire has completed all packet processing and the only task left to do is the attachment of a link layer header and the sending of the packet to the wire, then they do not need to check for the completeness status of the ire. Instead they can simply hand over the packet and the ire to ip_xmit_v4().

6.6 Optimizing the ICMP redirect work in FIB

In current Solaris, the entire FIB is traversed whenever the ICMP redirect timer (default value is 60 secs) goes off, even if there are no ires of type IRE_REDIRECT in the table. In a system, containing several hundred thousand entries in the FIB, these frequent, often unnecessary traversals cause dips in forwarding throughput.

Ideally a system set up as a pure router should ignore ICMP redirects. RFC 1812 (see reference 1), section 5.2.7.2 states:

"A router using a routing protocol (other than static routes) MUST NOT consider paths learned from ICMP Redirects when forwarding a packet. If

a router is not using a routing protocol, a router MAY have a configuration that, if set, allows the router to consider routes learned through ICMP Redirects when forwarding packets."

However, we expect our machine to be predominantly used as router *and* a host. To solve the problem of frequent unnecessary traversals of the forwarding table, we have added a global counter called *ip_redirect_cnt*, that keeps count of the IRE_REDIRECTS in the IPv4 FIB at all times. Whenever the redirect timer goes off, one can check for the value of this counter. A value of 0 will indicate that the traversal of FIB should be skipped, while a non-zero value will cause the traversal of the entire FIB.

7 New IP Filter API

The following API functions will allow simpification of IP Filter code. Specifically, it will allow removal of the following existing functions:

`ip_nexthop(), and ip_nexthop_route()`

as part of the upcoming pfhooks-api project. The specification of the API functions are as follows:

ifindex_lookup - Given a destination address, the API would supply the outgoing interface to use for sending a packet to this destination.The supplied dst_addr could be on-link host or off-link host.

```
/*
 * Return values: ifindex  or 0 (means failed)
 */
ifindex_lookup(const struct sockaddr *ipaddr, zoneid_t zoneid);
```

ipfil_sendpkt - The supplied ifindex may or may not be 0. IP will not manipulate ttl,checksumming, ipsec work for the data packet. IP Filter will handle fragmentation before calling this function. The supplied dst_addr could be on-link or off-link host.

```
/*
 *
 * Return values:
 * 0            IP was able to send of the data pkt
 * ECOMM        Could not send packet
 * ENONET       No route to dst.
 * EINPROGRESS  Transmission is being attempted though not guaranteed.
 *
 */
ipfil_sendpkt(const struct sockaddr *dst_addr, mblk_t *mp, uint_t ifindex,
    zoneid_t zoneid);
```

8 Future projects/RFEs

The following is a list of future projects or requests for enhancement (RFE) ideas:

- Implement full-blown ECMP.
- Improve the default route selection scheme to something better than current round-robin scheme.
- Extend insertion of incomplete ires in the host path.
- Merge ARP into IP.
- Polling implementation (Crossbow is slated to do this).
- Improvement of IPv6 forwarding scalability.

9 References

1. RFC 1812: *Requirements for IP Version 4 Routers*
2. *A Tree-Based Packet Routing Table for Berkeley Unix*, Author: Keith Skowler http://www.cs.berkeley.edu/~sklower/
3. *TCP/IP Illustrated, Volume 2, The Implementation*, Chapter 18: Radix Tree Routing Tables

4. *Fast Address Lookups using Controlled Prefix Expansion* Proc. ACM Sigmetrics '98 June 1998, p.1-11

5. *Eliminating Receive Livelock in an Interrupt-driven Kernel* Authors: Jeffrey Mogul, K.K. Ramakrishnan http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-8.html

10 Acknowledgements

The I-team would like to acknowledge the contributions and support of the following people (listed in alphabetical order):

- Adi Masputra
- Ashish Mehta
- Dan McDonald
- Darren Reed
- Erik Nordmark
- James Carlson
- Paul Jakma
- Paul Wernau
- Sunay Tripathi
- Thirumalai Srinivasan
- Venguopal Iyer

A High-level Code flow of IPv4 forwarding path

```
                            ip_input
                               |
              -------------------------------------------------
              |in a chain of incoming packets, is the previous|
              |packet's dst the same as the current one?      |
              -------------------------------------------------
                          |               | YES
                          | NO            | cache the ire
                          |               | obtained from last
                          |               | packet ("prev_ire")
                          ----------------
                                  |
               -------------------------------------
               |   CGTP, QOS, Mobile IP disabled    |
               |   and the packet is not multicast  |
               |   and not a IPPROTO_RSVP protocol  |
               |   packet?                          |
               -------------------------------------
                                |                  \
                                |                   \
                                |                    \
                                |                     \
                                | TRUE                  \ FALSE
                           -----------------              \
                        N- | prev_ire NULL ?|              |
                        |  -----------------               |
attach link-layer header|       | Y                        |
and send pkt on wire    | ire = ire_cache_lookup(dst)      | {FWD_SLOWPATH}
   ^                    |       |                          |
   ^                    `------>|                          |
   |                            |{ire of nexthop is NULL   |
   |{ire of nexthop complete    |or of IRE_CACHE type}     .
   | send it on the wire}       |                         /
   |                            |{FWD_FASTPATH}          /
   |                            |                       V
ire=ip_fast_forward()<-----------       ire = ip_rput_noire()
   |         |                             /
   |         |{ire of nexthop             /
   |         |is NULL}                   V
   |         `------------------>{FWD_SLOWPATH}
   |                     `>ire = ire_forward()
   |                     |   >ire_ftable_lookup() will return (ire, sire)
   |                     |      ire_t returned can be
```

```
   | {FWD_FASTPATH       |      1.IRE_CACHE of nxthop gateway
   |  Exception          |      2.IRE_IF_RESOLVER of nxthop gateway
   |  case: continue @   |      3.IRE_IF_NORESOLVER of nxthop gateway
   V [[JUNCTION F]]      |   >ire_create()
ip_rput_process_forward()|     > ire_init()
                         |        > ire_init_common()
                         |           > ire_nce_init()   [[JUNCTION B]]
                         |             > ndp_lookup_then_add()
                         |              {creates nce_t for IPv4 ires
                         |               with nce_state as "ND_REACHABLE"
                         |               for IRE_IF_NORESOLVER/IRE_BROADCAST
                         |               and "ND_INITIAL" for
                         |               IRE_IF_RESOLVER/ND_IRE_CACHEs
                         |               NOTE: IRE_LOOPBACK has no ire_nce}
                         |   > ire_add(new_ire,NULL,NULL,NULL)
                         |        [[JUNCTION A]]
                         |      {add incomplete ires in ctable}
                         |   > return newly created ire to
                         |     ip_rput_noire() which returns to ip_input()
                         |
                         |
                         ‘> ip_rput_process_forward() [[JUNCTION F]]
                            > ip_rput_forward()
                              > ip_xmit_v4()
                                 [[JUNCTION C]]
                                check ire->ire_nce->nce_state value
                                ire->ire_nce->nce_state can be one
                                of the following:
                                a) ND_INITIAL
                                b) ND_INCOMPLETE
                                c) ND_REACHABLE
                                For each case, continue as shown below:

 ND_INCOMPLETE                ND_INITIAL             ND_REACHABLE **
     |                            |                        |
     |                            |                        |
     |                            |                        V
     |                            |                        V
     ‘> nce_queue_mp_common       |                    dequeue each mp and:
     |                            |                    a)ip_wput_attach_llhdr()
     ‘> return status to          |                    b) if no IPsec hw accel:
        caller                    |                       putnext() - send it
                                  |                       on the wire
                                  |                       if IPsec hw accel
                                  |                       ipsec_hw_putnext()
```

```
                               `> nce_queue_mp_common()
                               |
                               `> ire_arpresolve()  [[JUNCTION D]]
                               |   create arp chain and send to ARP
                               |   (do *not* send the packet itself)
                               |   AR_ENTRY_QUERY -> "fake" ire_t w/db_type
                               |                     IRE_ARPRESOLVE_TYPE
                               |                                      $
                               `> change nce_state to ND_INCOMPLETE   $
                                                                      $
                                                                      $
                                                                     $
                                   (STREAMS msg from IP to ARP)     $
                                              $$$$$$$$$$$$$$$$$$$$$
                                    ARP MODULE
                                    send back reply from ar_query_reply();
                                    inserts xmit template after "fake" ire.
                                     (STREAMS msg from ARP to IP) $
                                             $$$$$$$$$$$$$$$$$$$$$
                                    answer is parsed at ip_wput()
[[JUNCTION E]]
ip_wput()->ip_output()->ip_wput_nondata()
                         |case (IRE_ARPRESOLVE_TYPE) :
                         `> ire = ire_ctable_lookup()
                         |
                         `> nce= ire->ire_nce
                         |
                         `> a) fill ire->ire->nce_res_mp
                         |     with the ARP response
                         `> b) nce->nce_state= ND_REACHABLE
                         |
                         `> c) ire_fastpath()
                         |
                          `> d)  ip_xmit_v4() **
                                refer to ND_REACHABLE case in
                                ip_xmit_v4() above
```

B Comparative analysis of FIB schemes

Test setup:

A Sun Fire V40Z quad 2.4GHz cpu AMD Opteron System was connected to the Tera Routing Tester from Spirent using the test setup shown below.

```
-----------------  data forwarded  ----------------------
|               |   -------->      | Smartbits SMB-6000  |
| Sun Fire V40Z |                  | Tera Routing Tester |
|               |   <--------      |  (version 4.5)      |
-----------------  data traffic    -----------------------
```

TRT 4.5 was used to generate routing tables of different sizes, which was uploaded into the V40Z using BGP. The TRT 4.5 interface was then used to generate packets at different rates to execute a throughput test as described in Section 3.17 of RFC 1242 .

Listed below is the comparative analysis of the current Solaris release and the prototypes of the two alternate schemes.

```
                     ***************************
                        EXISTING SOLARIS SCHEME
                     ***************************
```

TEST 1: Throughput and Memory usage

```
  --------------------------------------------------------
   Number of  route distribution   Thruput         Memory
    Routes                          (% of 1Gb/s)    (MB)
  --------------------------------------------------------
    1000        Internet                30.53        0.40
   10000        Internet                 8.75        3.75
  100000        Internet               <1.25        37.26
  150000        Internet               -            55.88
```

TEST 2: Forwarding throughput with route flap is not applicable since the throughput without route flap is so poor.

```
             ************************************
                EXISTING SOLARIS SCHEME (contd.)
             *************************************

TEST 3: Route add/delete times on Surya5:

Routes          Route add                Route flush
-------------------------------------------------------
smb100000       real     5m11.06s       real    1m5.20s
                user     0m6.17s        user    0m1.79s
                sys      0m16.65s        sys    0m4.31s

smb150000       real     9m5.76s        real    5m35.17s
                user     0m9.15s        user    0m5.05s
                 sys     0m24.56s        sys    0m27.61s
```

```
             **********************************************
                ADDRESS DIRECTED FORWARDING TABLE SCHEME
             **********************************************

TEST 1: Throughput and Memory usage
  --------------------------------------------------------
   Number of  route distribution   Thruput         Memory
    Routes                          (% of 1Gb/s)    (MB)
  --------------------------------------------------------
     1000        Internet        40.93              .48
    10000        Internet        40.93             4.26
   100000        Internet        40.93            42.06
   100000        Exponential     40.93            40.17
   112758        Even            40.93            56.39
   303217        Custom          40.93           142.60
  ----------------------------------------------------
```

- Space consumption depends on the route distribution, since the scheme works by partitioning the search space and not the set of keys (routes).
- In this implementation the worst case Trie memory needed is 944 bytes/route

TEST 2: Forwarding Throughput with Route Flap:
Test duration: 150 secs Flap schedule:

```
            Time Interval 30 sec
          Step 1: BGP Break TCP Session: ce: Session 2
            Time Interval 30 sec
          Step 2: BGP Restore TCP Session ce: Session 2
            Time Interval 30 sec
          Final Step: Unflap All
          Continuous flapping was disabled.

Note: All traffic was sent to session 1 (using Traffic
Wizard GUI) as Smartbits  will send traffic to session
2 even after the session is broken, and count the
(correctly) lost packets toward throughput measurement.

# of routes    Throughput
-------------------------
150000           40.93%
181892           40.93%
```

Max packet loss: 0.5

```
    ****************************************************
       ADDRESS DIRECTED FORWARDING TABLE SCHEME (contd)
    ****************************************************

TEST 3: Route add/delete times on Surya5:

Routes          Route add               Route flush
-------------------------------------------------------
smb100000       real     3:29.40        real        6.9
                user        6.3         user        0.2
                sys        16.8         sys         2.4

smb150000       real     5:13.2         real        4.0
                user        9.4         user        0.3
                sys        24.1         sys         3.6


                     ************************
                        RADIX TREE SCHEME
                     ************************

TEST 1: Throughput and Memory usage
  ---------------------------------------------------------
   Number of  route distribution   Thruput        Memory
    Routes                         (% of 1Gb/s)    (MB)
  ---------------------------------------------------------
   1000         Internet            40.93          0.53
  10000         Internet            40.93          5.29
 100000         Internet            38.96         52.95
 100000         Exponential         39.53         52.98
 112764         Even                38.66         35.50
 303217         Custom              41.38        160.67
------------------------------------------------------------
```

```
                  ****************************
                    RADIX TREE SCHEME (contd)
                  ****************************

TEST 2: Forwarding Throughput with Route Flap:

Test duration: 150 secs
Flap schedule:

            Time Interval 30 sec
          Step 1: BGP Break TCP Session: ce: Session 2
            Time Interval 30 sec
          Step 2: BGP Restore TCP Session ce: Session 2
            Time Interval 30 sec
          Final Step: Unflap All
          Continuous flapping was disabled.
Note: All traffic was sent to session 1 (using Traffic
Wizard GUI) as Smartbits  will send traffic to session
2 even after the session is broken, and count the
(correctly) lost packets toward throughput measurement.

# of routes    Throughput
-------------------------
150000             39.41%
181892             37.01%

Max packet loss: 0.5%

TEST 3: Route add/delete times on Surya5:
Route add/delete times on Surya5:

Routes          Route add               Route flush
-------------------------------------------------------
smb100000       real    3m27.06s        real    1m8.50s
                user     0m7.92s        user    0m1.97s
                sys     0m14.72s        sys     0m5.26s

smb150000       real     5m2.37s        real   5m29.06s
                user    0m11.95s        user    0m4.96s
                sys     0m21.12s        sys    0m23.86s
```