Optimizing the BSD Routing System for Parallel Processing

Qing Li
Blue Coat Systems, Inc.
Sunnyvale, CA, USA
qing.li@bluecoat.com,
qingli@freebsd.org

Kip Macy
The FreeBSD Project
Palo Alto, CA, USA
kmacy@freebsd.org

ABSTRACT

The routing architecture of the original 4.4BSD [3] kernel has been deployed successfully without major design modification for over 15 years. In the unified routing architecture, layer-3 (L3) IP routes are maintained with layer-2 (L2) ARP entries in the same kernel structures. This meant that a single table lookup can return both results. Today, the prevalence of multi-core CPUs and parallel processor architectures is driving the re-design of software data structures and control flows to fully exploit the parallel capabilities of commodity hardware. A common parallel TCP/IP network protocol stack design separates out L2 and L3 processing from layer-4 (L4) and layer-5 (L5) (TCP and socket) onto different CPU cores. The unified routing architecture creates data dependencies between these layers, complicating the design and causing high levels of lock contention. In this paper we will detail the routing architecture that we have implemented for the upcoming FreeBSD 8.0 kernel, which eliminates the data dependencies and facilitates better parallelization of the network protocol stacks. We will describe the impact of this design on higher layer protocols such as TCP and UDP flow processing, and provide performance comparison between the original and the new design.

Categories and Subject Descriptors

C.2.5 [**Local and Wide-Area Networks**]: Ethernet, Internet; C.2.6 [**Internetworking**]: Routers; D.4.1 [**Operating Systems**]: Process Management – concurrency, mutual exclusion, synchronization; D.4.4 [**Communications Management**]: Network Communication

General Terms

Algorithms, Design, Performance

Keywords

FreeBSD, synchronization, server load balancing (SLB), flow table, IP, IPv6, ARP, Neighbor Cache, SMP, MP, routing

PRESTO'09, August 21, 2009, Barcelona, Spain.

1. INTRODUCTION

Much of the original 4.4BSD operating system design, including the unified routing architecture, can be found in the NetBSD and OpenBSD operating systems, and in FreeBSD up through version 7. These modern 4.4BSD descendants have been widely adopted by industry and have been commercialized into various networking and router products. In this paper our discussion is centered around the FreeBSD operating system. The FreeBSD networking kernel is an integral part of Wind River's VxWorks 6.x, Cisco's Ironport products, Apple OS X, and Juniper's routers. In addition, FreeBSD is constantly being ported to new custom hardware platforms.

The routing architecture of the original 4.4BSD kernel was not modified in the FreeBSD OS for over 15 years. Although the unified routing architecture is still present in most modern descendants of 4.4BSD, in this paper our focus is on FreeBSD 8.0, in which it has been replaced. Thus we refer to it as the "legacy unified design." In the legacy unified design, the L3 IP routes were maintained with the L2 ARP entries in the same kernel data structures. Similarly, the IPv6 routes were maintained with the Neighbor Cache (ND6) [2] entries. This design, although somewhat more complex, provided the benefit of reducing lookups by letting the caller retrieve both the interface information and the L2 information in one function call. With the move from Uni-Processor (UP) to Multi-Processor (MP), this design proved to be problematic in several areas. Those products that are based on FreeBSD inherited those design limitations unless those limitations were resolved through custom design in the derived products.

The main goals for redesigning the kernel routing infrastructure was to reduce the scope of the customization necessary when deriving products from FreeBSD, and to offer a generic solution that could be an integral part of the kernel. The FreeBSD community can now evolve and improve this general solution over time. In this paper IP or L3 refers to both IPv4 and IPv6, and the L2 entries refer to both ARP and ND6 entries.

1.1 Route Cloning and L2 Entry Creation and Deletion

In the legacy unified routing design, each time an IP address was assigned to an interface, a prefix route (or interface route) was installed into the kernel routing table. This prefix route had a property named *route cloning capable.* All directly reachable nodes of that prefix would be *cloned* from the prefix route to store the L2 addresses

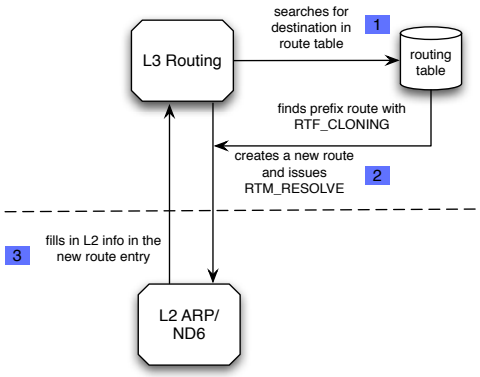

Figure 1: L2 route entry creation

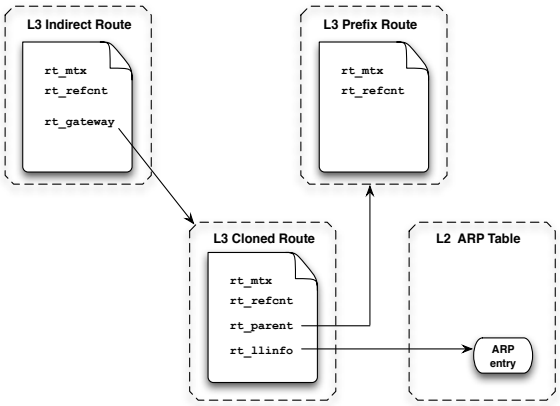

Figure 2: legacy route types and inter-relations

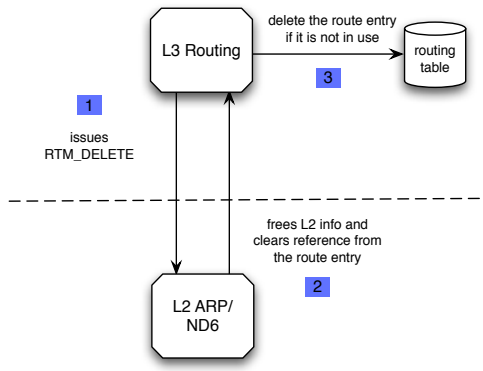

Figure 3: L2 route entry deletion

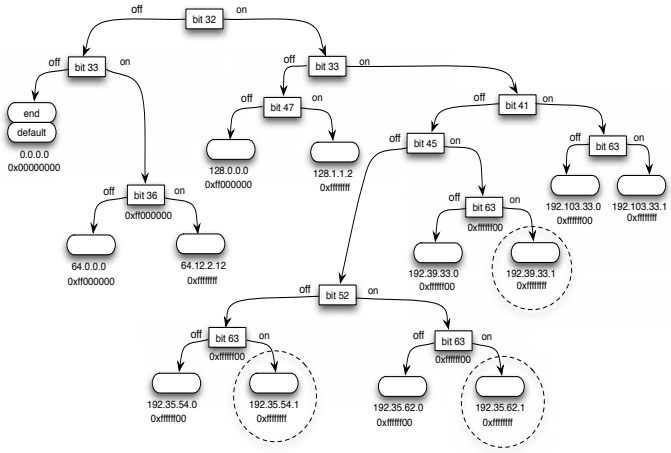

Figure 4: example radix tree

of those nodes. Figure 1 illustrates the cloning process. When creating an L2 entry, L3 allocated a routing table entry, `rtentry`, and inserted it into the routing table. The new `rtentry`'s `rt_parent` pointer would be set to point at the prefix route's `rtentry`. In the event that the new `rtentry` contained the link address for a gateway, the next time any `rtentry` of an indirect route using that gateway was accessed, the `rt_gateway` pointer in the indirect route's `rtentry` would be set to point at the new cloned route. Then L3 issued a `RTM_RESOLVE` request to L2 to populate this new `rtentry` with the corresponding L2 address. L2 performed the appropriate address resolution according to the underlying network type, allocated a memory block to hold the L2 address, and then initialized the `rt_llinfo` field of the route entry to point to this memory block. Figure 2 depicts the relationship among the various types of route entries.

When deleting an L2 entry, L3 issued a `RTM_DELETE` request to L2. L2 freed the memory block associated with the L2 address and set the `rt_llinfo` pointer to NULL. L3 then freed the `rtentry` and removed it from the routing table if it was not currently in use by any other threads.

Figure 3 illustrates the cloned `rtentry` deletion process.

When deleting a prefix route, the routing table would first need to be walked, deleting all `rtentrys` whose `rt_parent` pointer pointed at the prefix route's `rtentry`. Only then could the prefix route's `rtentry` be removed from the table and freed.

Figure 4 illustrates an example routing table that includes both prefix routes and ARP entries [4]. The cloned routes (or ARP entries) are represented by those radix tree nodes that are enclosed in circles. In this example, the routing table represents a system with five interfaces that are connected to five different IP networks. It is easy to see from Figure 4 that when searching for an ARP entry, for example for 192.35.62.1, unrelated nodes (of different prefixes) needed to be visited before reaching the entry for 192.35.62.1.

1.2 Limitations of the Legacy Unified Design

Each `rtentry` has a reference count `rt_refcnt` which is guarded by a lock (`rt_mtx`). When a `rtentry` is in use, the `rt_refcnt` field has a non-zero value. The `rt_mtx` lock must be held before updating the `rt_refcnt`. This lock may be acquired and released multiple times throughout the packet processing path. For example, for an indirect route, the route's `rtentry` was retrieved first to determine the first-hop router. The `RTF_GATEWAY` flag had special significance to L2, e.g. the ARP resolution function `arpresolve()`, indicating that the `rt_gwroute` field, if non-NULL, contained a valid pointer to the L2 `rtentry` for the gateway. Although validating `rt_llinfo` was a simple task, the coupling of L3 and L2 `rtentrys` for gateway routes made the control flow and locking in `rt_check()`, the function for validating `rt_llinfo`

in L2, extremely complicated and error-prone. One of the main reasons for this locking complexity was the fact that the L2 address that is associated with a cloned route may be freed while references to the `rtentry` are held.

For example, in the FreeBSD 5.4 release, a TCP connection cached a reference to the `rtentry` that was used to transmit the initial TCP `SYN` packet. This cached `rtentry` was then passed from TCP to IP to avoid further routing lookup if it were still marked as valid. Although the `rt_refcnt` is non-zero for this cached `rtentry`, indicating that it is still live, the L2 address pointed at by `rt_llinfo` may already have been freed by another thread. Thus, when the L2 address that was stored in `rt_llinfo` is accessed, the `rt_mtx` lock must be held to guarantee the liveness of `rt_llinfo`. If the L2 address was invalid, then the current `rt_mtx` lock is released, and the `RTM_RESOLVE` process was repeated to reinitialize `rt_llinfo`.

Consequently, even though the L3 state of the `rtentry` could be safely accessed with only a reference held, its `rt_mtx` lock needed to be held to safely access the `rt_llinfo` L2 information. Although the `rtentry` itself can be cached by reference counting to avoid subsequent lookups and lock acquisitions, accessing the L2 information required the acquisition of the `rt_mtx` lock and possibly incurred additional lookup cost.

The legacy design reduced parallelism on SMP and parallel architectures. As a result of the dependency between L2 and L3, the processing through these two layers was single threaded. A common parallel TCP/IP protocol stack design is to allow L2 and higher layer processing to run independendently of each other, having each processor managing different protocols. The aforementioned locking contention increased processor stalling and prevented one from benefitting from more advanced hardware platforms.

This processor stalling effect was especially apparent when the majority of the connections were going through the default router and the cloned `rtentry` held the L2 address of the default gateway. Cache invalidation increases with higher frequency of lock acquisitions and lock releases.

The cross layering data dependency also prevented full utilization of the underlying capability in advanced platforms where L2 processing is offloaded to hardware. The FreeBSD networking kernel has been ported to run on custom hardware platforms where the L2 engine has the ability to manage ARP table in ASICs. Again, the existing design complicates any solution aimed at achieving good parallelism.

Hardware vendors such as Cavium Networks® understand the synchronization issues related to the ARP table and route table management in the legacy system. As such these vendors provide customized implementation for their specialized core to solve some of these scalability problems. However, these custom solutions are only available commercially.

2. OVERVIEW OF THE NEW SPLIT ROUTING DESIGN

In the legacy unified design the L2 `rtentry`s were created out of the prefix route `rtentry`s. The only difference between a prefix route `rtentry` and a cloned L2 route `rtentry` was that a L2 route `rtentry` held a reference to the link-layer address. The link-layer address was not accessed by L3 but was copied into the packet header as the source MAC address inside L2. In contrast to route cloning, with the split design, the IP destination (that is either the first-hop router or the final packet destination) can be given to L2 along with the output packet. L2 can determine if address resolution is necessary for the given IP destination, and if so performs that task. The result is then maintained inside L2 rather than L3. The L3 routing table is now dedicated to prefix routes, default routes, and indirect routes (for off-link nodes).

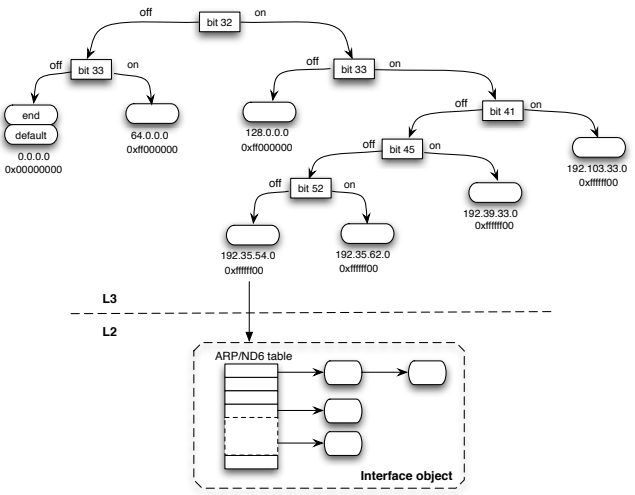

Figure 5: split L3 and L2 tables

Figure 5 illustrates the structures of the new routing and L2 tables, and the relationship between the two tables. In the split design, when communicating with on-link nodes, only the prefix route `rtentry` is retrieved from the routing table to obtain the interface information. Then the destination IP is passed to L2.

In the legacy design, communicating with an off-link node required manipulating two `rtentry`s. In the new design only one `rtentry` is examined. This comes at the expense of an additional search in the L2 table of the selected interface. We demonstrate in Section 3 that this extra search adds no measurable overhead. Furthermore, in Section 4, we demonstrate that the separate reference counting of L2 information can provide remarkable scalability benefits.

As shown in Figure 5, instead of being centralized in the routing table, each interface maintains an ARP table if the IP protocol is registered on the interface. Similarly, each interface maintains an ND6 table if the IPv6 protocol is registered on the interface. Each L2 table contains L2 entries for those nodes that are directly reachable over that interface. The per-interface L2 table is currently implemented as a hash table. Other protocols can register their own L2 tables with the interface.

Comparing Figure 1 and Figure 5, one can see the new routing table is much smaller than the legacy routing table. The reduction in routing table size means fewer radix nodes are traversed during a search, which can result in fewer data cache invalidations.

By eliminating the routing table dependency between L2 and L3, the routing table lock hold time is reduced, which also reduces lock contention. Once a packet is transferred from L3 to L2, L3 can continue to process additional packets

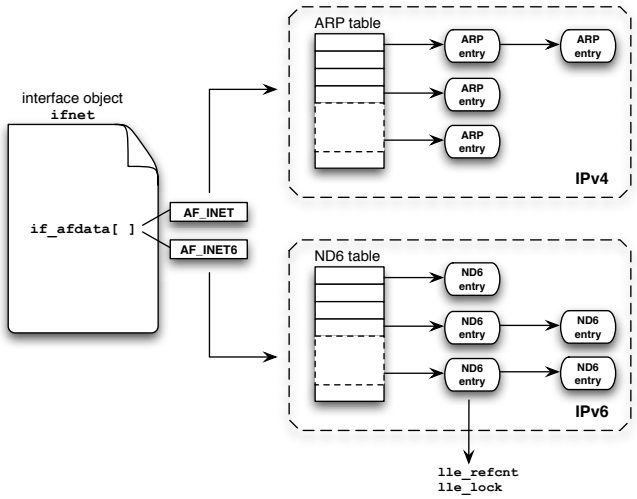

Figure 6: new L2 table structure

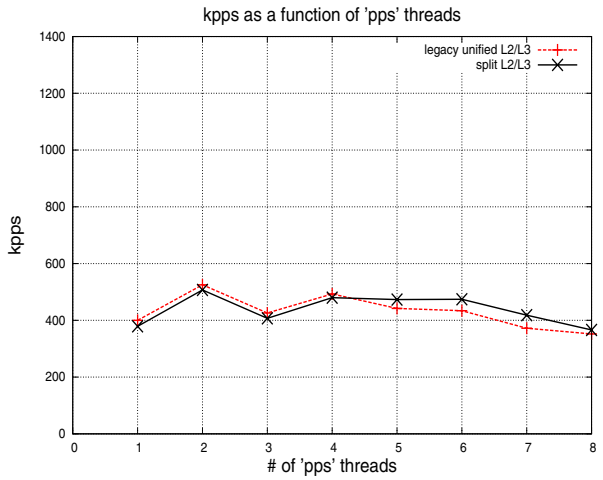

Figure 7: Throughput Comparison for Split L2/L3

without having to wait for L2 completion. L2 packet processing is independent of L3 activities thus increasing the parallelism in packet processing.

2.1 Reducing Locking Complexity

Since the L2 tables are now maintained in the per interface `ifnet` object, fine grained locking can be used to serialize access to the L2 tables. As shown in Figure 6, the IPv4 ARP and IPv6 ND6 tables are maintained in the per *address family* `if_afdata[]` array. Access to either table is synchronized with the `if_afdata_lock` read-write lock.

Another serious problem with the legacy unified design was that the L2 layer could update a `rtentry` without L3 involvement. This causes the cache line holding the `rtentry` to be marked dirty causing coherency overhead for both the lock and the `rt_llinfo` field.

In the new architecture each link-layer entry (LLE) is protected by a reference count `lle_refcnt` and a read-write lock `lle_lock`. The L2 `LLE` object is maintained and managed by L2 only. L3 can cache a L2 `LLE` reference, and any change to the `LLE` object can be detected without locking by L3 the next time the object is referenced.

2.2 Code Reduction

Another notable benefit of the new routing design was a 30% code reduction in both the L2 and the L3 modules.

All of the logic related to route cloning was completely removed. For example, it is no longer necessary to issue the `RTM_RESOLVE` call between L2 and L3. Since L2 no longer references L3 `rtentry`s, the `rtentry` validation routine, `rt_check()` was made obsolete.

The routing infrastructure code is much easier to understand in the new design. Consequently, the reduction in complexity makes the code less prone to programming errors.

2.3 APIs and Compatibility

The FreeBSD OS comes with thousands of ported applications. There were a number of userland applications that were affected by the redesign in the kernel routing infrastructure. Currently these applications interact with the kernel routing infrastructure through the socket interface by means of routing socket messages. In the legacy unified routing table cloned `rtentry`s holding an L2 address were marked with the `RTF_LLINFO` flag. Applications such as `netstat` that display the routing tables issue a single command to retrieve an entire table. These applications assume the routing table contains both L3 and L2 information, therefore they have built-in filter code that looks for the `RTF_LLINFO` flag to remove ARP entries, and display only L3 information. This filter code is now redundant, but the existence of such code has no effect on the actual operation.

Applications such as `arp` and `ndp` that interact with the L2 tables, for example, adding or deleting ARP or ND6 entries, also specify the `RTF_LLINFO` flag inside the routing socket message. Although the route cloning concept is obsolete, the kernel continues to process this type of message to ensure application binary compatibility.

Applications such as `net-snmp` may monitor routing table update messages. In principle `net-snmp` may also be setup to monitor L2 table update messages. When a change takes place in either the ARP or NDP table, the kernel will generate an update notification to the listening applications with a L2 route entry marked with the `RTF_LLINFO` flag.

3. PERFORMANCE COMPARISON

For our performance testing we used a system containing two Intel® Xeon® L5420 CPUs running at 2.50GHz for a total of eight cores. To measure packet throughput we used a simple custom benchmark, **pps**, that sends UDP packets as quickly as the interface will allow. We took measurements for one to eight sending threads, the average of five fifteen second samples was used for each data point.

Figure 7 contrasts legacy and split L2/L3 packet sending performance. The added lookup in the interface's L2 table is masked by the overhead of `rtentry` locking for `rt_refcount` changes in the IP packet output processing function `ip_output()`. One can see that throughput does not increase as the number of sending threads increases.

Although disappointing at first glance, the main bottleneck of the locking protecting reference count changes to `rtentry`s in function `ip_output()` still exists and needs to be addressed separately.

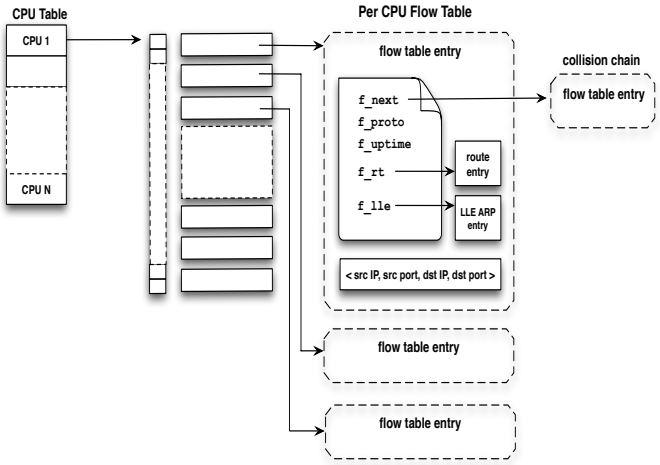

Figure 8: Flow Table Structure

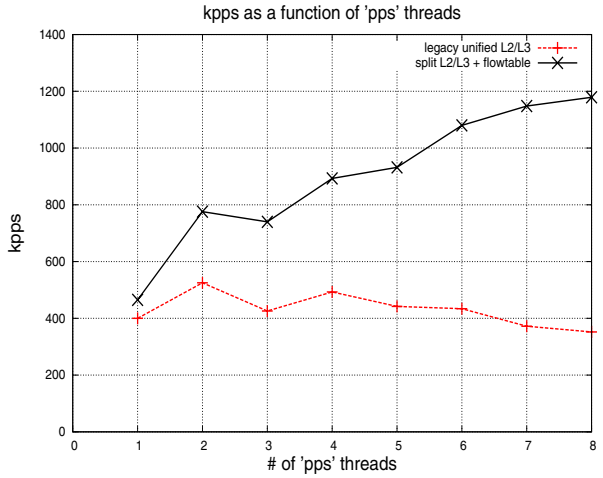

Figure 9: Throughput Comparison for Added Flow Table

4. SPLIT L2/L3 DESIGN AS A FOUNDATION FOR OTHER FUNCTIONALITY

In April 2008, equal-cost multi-path (**ECMP**) support was integrated into FreeBSD. ECMP allows multiple routes to have the same destination prefix but different first-hop gateways. This enables FreeBSD to uniformly distribute connections matching the same destination prefix across multiple routes.

In the initial implementation, a first-hop `rtentry` was chosen by using the result of the bitwise XOR of the source and the destination IP addresses for a packet modulo the number of next-hop `rtentry`s as an index into the prefix route's `rtentry` list. Although this works well for the general forwarding case, its statelessness proves to be problematic when connections terminate at the next hop, as is the case with Server Load Balancing (**SLB**) [1]. When additional servers are added to, or removed from, the prefix list, the modulus used to calculate the `rtentry` chosen will change and all packets for a given source/destination pair will be directed to a different server. The new server will have no knowledge of these connections and will send a TCP `RST` packet to the client, telling it to terminate the connection.

4.1 Flow Table for Stateful Forwarding

In order for FreeBSD to be used for SLB with long-lived TCP connections, forwarding information must be stateful. To this end, a flow table was added. In essence, the flow table hashes the connection information to a flow entry, `flentry`, which contains a reference to a `rtentry` and a `LLE`.

Prior to the implementation of the split L2/L3 design, it was not possible to efficiently implement a flow table. The flow table has two modes, hashing the 4-tuple and protocol value and hashing just the destination address. A `flentry` is retained until it has not been used for more than a configurable timeout number of seconds, by default 30. This is tracked by a field in the entry that is updated each time it is accessed with the kernel's system *uptime*. The flow table is managed on a per-CPU basis, avoiding any lock contention and cache coherency overhead after the initial lookup.

Figure 8 shows the flow table structure. For more efficient garbage collection of stale `flentry`s a bitmap of used entries is also kept per-cpu. A hash table entry contains at least one flow table entry if its bit field is set. This greatly reduces the number of buckets that need to be checked in the sparse hash table case.

4.2 Flow Table Performance

Although the primary purpose of the flow table is to provide stateful forwarding for multipath use in load balancing, it can also serve to bypass L3 and L2 lookups for recently seen destinations in general. A call to `flowtable_lookup()` was added to `ip_output()`, this uses the flow table in the non-load-balancing mode, hashing the destination IP. The cached L3 entry supplies the output interface and the cached L2 entry supplies the destination MAC address.

To measure the performance benefits of the flow table we ran the same **pps** benchmark with the flow table enabled. Figure 9 shows the results. After adding caching of the L2 and L3 entries using the flow table the system spends less than 10% of system cpu time on locking operations with 8 sending threads. Greater than 30% of system cpu time is spent in the **em** Intel® gigabit ethernet device driver. This leads us to conclude that scaling is limited by the **em** device driver and not by the network stack.

We also measure the locking overhead of the unified design as compared with the flow table. The locking overhead measurements were done by sampling the unhalted core cycles for thirty seconds using FreeBSD's **pmc** performance counters tools. The percentage time taken by all locking operations consuming more than 1% of cpu time were added to calculate the total locking overhead. Figure 10 shows the results of measuring the locking overhead. The '0' measurement was taken by sampling time spent in locking operations on an otherwise idle system in multi-user mode.

In the legacy unified L2/L3 design contention on locks was inevitably a major portion of cpu time, reaching as high as 47% with 8 transmitting threads. With the new split L2/L3 design the L2 and L3 references can be cached in the protocol control block for connected sockets or in a flow table for unconnected sockets and forwarding. Thus we see that very little of the cpu time is now spent in the locking primitives even when there are 8 transmitting threads.

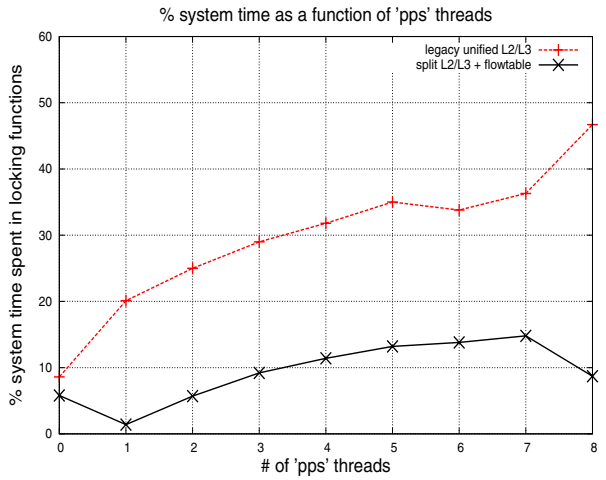

Figure 10: Locking Overhead Comparison

There are clear limitations to the flow table approach to bypassing locking in some environments. When doing forwarding to more than a million unique destination IPs, the actively referenced `flentry`s will no longer fit in the system's L2 cache. This would have the effect of requiring at least one un-prefetchable memory access per packet. We briefly discuss how to address this issue in the next section.

5. FUTURE WORK

With the advent of the split L2/L3 design there is the potential for future further improvements to the routing table to change lookup semantics to eliminate the need for reference counting `rtentry`s. Upon a successful lookup of an rtentry, its contents could be copied instead of locking the `rt_mtx`, incrementing `rt_refcount`, and then returning a pointer to the `rtentry`. This would make possible higher performance forwarding by allowing the members of the routing table to sit in L2 cache in a shared state because there would be no need for `rt_refcount` or `rt_mtx`, the update and acquisition of which require cache lines to be evicted. This would require changes to the flow table to regularly validate cached `rtentry`s but this would be largely offset by eliminating the overhead of serializing `rtentry` reference count manipulation.

The flow table currently only supports IPv4 and is statically sized at boot time. In the future we will extend it to add IPv6 and be dynamically re-sizable at run-time. Another interesting extension to ECMP and the flow table would be to support per-packet load balancing to increase the amount of bandwidth available to individual connections. Load balancer failover support could be made possible by adding an interface to export and import the flow table state to and from user processes to allow the table to be mirrored between multiple systems.

6. CONCLUSIONS

In this paper we have described in detail both the unified L2/L3 routing infrastructure and the new split L2/L3 architecture that we have implemented in the FreeBSD 8.0 kernel. We have shown the new design provides the following benefits: simplified locking and reduced lock contention; elimination of data dependence between L2 and L3 - facilitating SMP [5] and parallel design; improved data cache utilization and the possibility of L2 hardware offload. As a use case we showed that by adding a flow table we could provide stateful forwarding to ECMP and bypass `rtentry` and `LLE` locking in the packet output path when processing established flows.

7. REFERENCES

[1] K. Chandra. *Load Balancing, Servers, Firewalls, and Caches*. John Wiley & Sons, Inc., 2002.

[2] S. K. Li Q., Jinmei T. *IPv6 Core Protocols Implementation*. Morgan Kaufmann, 2006.

[3] N. N. G. McKusick M. *The Design and Implementation of the FreeBSD Operating System*. Addison Wesley Longman, 2004.

[4] S. R. W. Wright Gary R. *TCP/IP Illustrated, Volume 2, The Implementation*. Addison Wesley Longman, 1995.

[5] R. N. M. Watson. Introduction to multithreading and multiprocessing in the freebsd smpng network stack. In *Proceedings of EuroBSDCon 2005*, November 2005.

8. ACKNOWLEDGMENTS

Qing Li would like to thank Blue Coat Systems for sponsoring the implementation work, and for granting him the publication rights. Kip Macy would like to thank BitGravity for supporting the development of the flow table and for providing the resources for doing the performance testing. We would like to thank the various FreeBSD developers who have contributed to the new routing design, Sam Leffler, Andre Oppermann, and Luigi Rizzo.