Anatomy of a Linux bridge

Nuutti Varis
Aalto University School of Electrical Engineering,
Department of Communications and Networking
P.O.Box 13000, 00076 Aalto, Finland
Email: {firstname.lastname}@aalto.fi

ABSTRACT

Ethernet is the prevalent Local Area Networking (LAN) technology, offering a cost efficient way to connect end-hosts to each other. Local area networks are built by networking devices called switches, that forward Ethernet frames between end-hosts in the network. The GNU/Linux operating system can be used to create a software based switch, called a *bridge*. This paper explores the architecture, design, and implementation of the Linux bridging component, and attempts to chart some of the processing characteristics of the frame forwarding operation, inside the bridge and in the operating system as a whole.

1. INTRODUCTION

Network devices, called switches (or synonymously, *bridges*) are responsible for connecting several network links to each other, creating a local area network. Conceptually, the major components of a network switch are a set of network ports, a control plane, a forwarding plane, and a MAC learning database. The set of ports are used to forward traffic between other switches and end-hosts in the network. The control plane of a switch is typically used to run the Spanning Tree Protocol (STP) [15], that calculates a minimum spanning tree for the local area network, preventing physical loops from crashing the network. The forwarding plane is responsible for processing input frames from the network ports, and making a forwarding decision on which network ports the input frame is forwarded to.

Finally, the MAC learning database is used to keep track of the host locations in the LAN. It typically contains an entry for each host MAC address that traverses the switch, and the input port where the frame was received. The forwarding decision is based on this information. For each unicast destination MAC address, the switch looks up the output port in the MAC database. If an entry is found, the frame is forwarded through the port further into the network. If an entry is not found, the frame is instead flooded from all other network ports in the switch, except the port where the frame was received. This latter provision is required to guarantee the "plug-and-play" nature of Ethernet.

In addition to Linux, several other operating systems also implement local area network bridging in the network stack. FreeBSD has a similar bridging implementation to Linux kernel, however the FreeBSD implementation also implements the Rapid Spanning Tree Protocol (RSTP). The FreeBSD bridge implementation also supports more advanced features, such as port MAC address limits, and SNMP monitoring of the bridge state. OpenSolaris also implements a bridging subsystem [12] that supports STP, RSTP, or a next generation bridging protocol called TRILL [14].

There has been relatively little evolution in bridging since the inception of the STP. Switches have evolved in conjunction with other local area network technologies such as Virtual LANs [16], while the STP has been incrementally extended to support these new technologies. Currently, there are two practical next-generation solutions for switching: RBridges (TRILL), and the Shortest Path Bridging (SPB) [1]. Both TRILL and SPB diverge from STP based bridging in several important ways. Some of the key differences are improved loop safety, more efficient unicast forwarding, and improved multicast forwarding. Additionally, the well known scalability issues [2] of the local area networks, and the advent of data center networking has also created a number of academic research papers, such as SPAIN [10], Port Land [11], VL2 [6], DCell [7], and BCube [8].

This paper explores the architecture, design and the implementation of the Linux bridging module. In addition, the paper also analyzes the processing characteristics of the Linux bridging module by profiling the kernel during forwarding, and observing various counters that track the performance of the processors and the memory in the multi-core CPU. The design and implementation of STP in the Linux bridge module is considered out of scope for this paper.

The rest of the paper is structured as follows. Section 2 presents an overview of the central data structures of the Linux bridge, creation of a Linux bridge instance, and the processing flow of an incoming frame. Next, Section 3 describes the forwarding database functionality of the bridge implementation. Section 4 describes the experimentation setup, and analyzes some of the performance related aspects of the bridging module and the operating system. Finally, Section 5 finishes the paper with some general remarks of local area networks and the Linux bridging implementation.

2. OVERVIEW

The architectural overview of the Linux bridging module is divided into three parts. First, the key data structures for the bridging module are described in detail. Next, the configuration interface of the Linux bridging module is discussed by looking at the bridge creation and port addition mechanisms. Finally, the input/output processing flow of

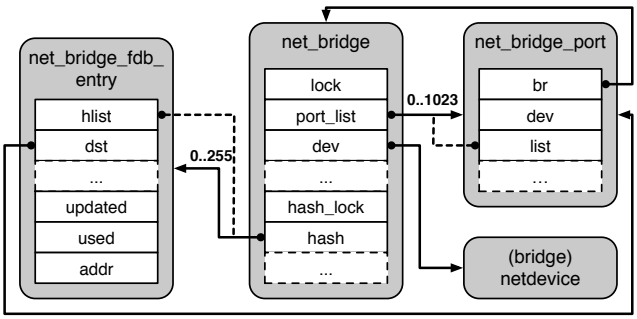

Figure 1: Primary Linux bridge data structures

the Linux bridging module is discussed in detail.

2.1 Data structures

The Linux bridge module has three key data structures that provide the central functionality for the bridge operation. Figure 1 presents an overview of the most important fields and their associations in the three key data structures. The main data structure for each bridge in the operating system is the `net_bridge`. It holds all of the bridge-wide configuration information, a doubly-linked list of bridge ports (`net_bridge_port` objects) in the field `port_list`, a pointer to the bridge netdevice in the field `dev`, and the forwarding database in the field `hash`. The technical details and the functionality of the hash array table are described in 3.1. Finally, the field `lock` is used by the bridge to synchronize configuration changes, such as port additions, removals, or changing the various bridge-specific parameters.

Each bridge port has a separate data structure `net_bridge_port`, that contains the bridge port specific parameters. The field `br` has a back reference to the bridge that the port belongs to. Next, the `dev` field holds the actual network interface that the bridge port uses to receive and transmit frames. Finally, position of the data structure object in the `net_bridge->port_list` linked list is stored in the field `list`. There are also various configuration parameter fields for the port, as well as the port-specific state and timers for the STP and IGMP [5] snooping features. IGMP snooping will be detailed in Section 3.2.

Finally, the third key data structure for the Linux bridge module is the `net_bridge_fdb_entry` object that represents a single forwarding table entry. A forwarding table entry consists of a MAC address of the host (in the field `addr`), and the port where the MAC address was last seen (in the field `dst`). The data structure also contains a field (`hlist`) that points back to the position of the object in a hash table array element in `net_bridge->hash`. In addition, there are two fields, `updated` and `used`, that are used for timekeeping. The former specifies the last time when the host was seen by this bridge, and the latter specifies the last time when the object was used in a forwarding decision. The `updated` field is used to delete entries from the forwarding database, when the maximum inactivity timeout value for the bridge is reached, i.e., $current\_time - updated > bridge\_hold\_time$.

2.2 Configuration subsystem

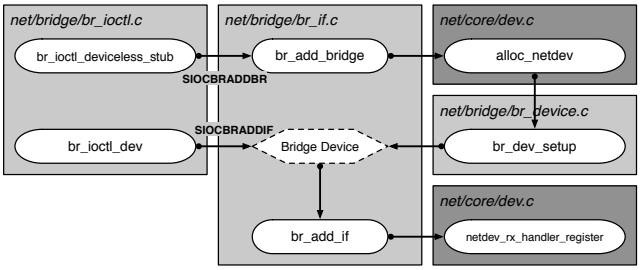

Figure 2: Linux bridge configuration; adding a bridge and a bridge port

The Linux bridging module has two separate configuration interfaces exposed to the user-space of the operating system. The first, ioctl interface offers an interface that can be used to create and destroy bridges in the operating system, and to add and remove existing network interfaces to/from the bridge. The second, sysfs based interface allows the management of bridge and bridge port specific parameters. Figure 2 presents a high level overview of the kernel ioctl process, that creates and initializes the bridge object, and adds network interfaces to it. The functions on dark grey areas are in the generic kernel, while the lighter areas are in the bridge.

The creation of a new bridge begins with the ioctl command *SIOCBRADDBR* that takes the bridge interface name as a parameter. The ioctl command is handled by the `br_ioctl_deviceless_stub` function, as there is no bridge device to attach the ioctl handler internally. The addition of a new bridge calls the function `br_add_bridge`, that creates the required bridge objects in the kernel, and eventually calls the `alloc_netdev` function to create a new netdevice for the bridge. The allocated netdevice is then initialized by the `br_dev_setup` call, including assigning the bridge device specific ioctl handler `br_dev_ioctl` to the newly allocated netdevice. All subsequent bridge specific ioctl calls are done on the newly created bridge device object in the kernel.

Ports are added to bridges by the ioctl command *SIOCBRADDIF*. The ioctl command takes the bridge device and the index of the interface to add to the bridge as parameters. The ioctl calls the bridge device ioctl handler (`br_dev_ioctl`), that in turn calls the `br_add_if` function. The function is responsible for creating and setting up a new bridge port by allocating a new `net_bridge_port` object. The object initialization process automatically sets the interface to receive all traffic, adds the network interface address for the bridge port to the forwarding database as a local entry, and attaches the interface as a slave to the bridge device. Finally, the function also calls the `netdev_rx_handler_register` function that sets the `rx_handler` of the network interface to `br_handle_frame`, that enables the interface to start processing incoming frames as a part of the bridge.

2.3 Frame processing

The Linux bridge processing flow begins from lower layers. As mentioned above, each network interface that acts as a bridge interface, will have a `rx_handler` set to `br_handle_frame`, that acts as the entry point to the bridge frame processing code. Concretely, the `rx_handler` is called by

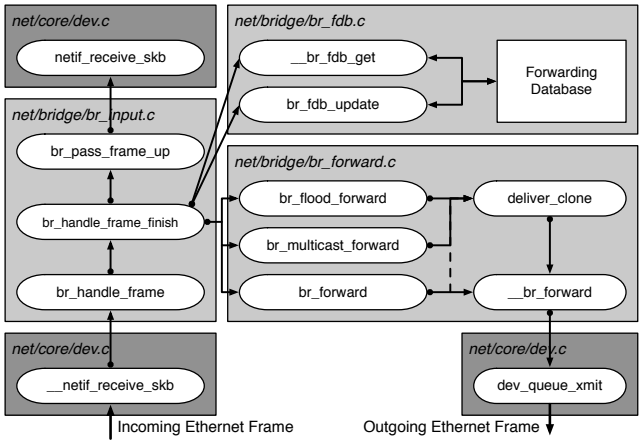

Figure 3: Architectural overview of the Linux bridge module I/O

the device-independent network interface code, in `__netif_receive_skb`. Figure 3 presents the processing flow of an incoming frame, as it passes through the Linux bridge module to a destination network interface queue.

The `br_handle_frame` function does the initial processing on the incoming frame. This includes doing initial validity checks on the frame, and separating control frames from normal traffic, because typically these frames are not forwarded in local area networks. The bridge considers any frame that has a destination address prefix of `01:80:C2:00:00` to be a control frame, that may need specialized processing. The last byte of the destination MAC address defines the behavior of the link local processing. Currently, Ethernet pause frames are automatically dropped, STP frames are either passed to the upper layers if it is enabled on the bridge, or forwarded, when it is disabled. Finally, if a forwarding decision is made, and the bridge is in either forwarding or learning mode, the frame is passed to `br_handle_frame_finish`, where the actual forwarding processing begins.

The `br_handle_frame_finish` function first updates the forwarding database of the bridge with the source MAC address, and the source interface of the frame by calling `br_fdb_update` function. The update either inserts a new entry into the forwarding database, or updates an existing entry.

Next, the processing behavior is decided based on the destination MAC address in the Ethernet frame. Unicast frames will have the forwarding database indexed with the destination address by using the `__br_fdb_get` function to find out the destination `net_bridge_port` where the frame will be forwarded to. If a `net_bridge_fdb_entry` object is found, the frame will be directly forwarded through the destination interface by the `br_forward` function. If no entry is found for the unicast destination Ethernet address, or the destination address is broadcast, the processing will call the `br_flood_forward` function. Finally, if the frame is a multi-destination frame, the multicast forwarding database is indexed with the complete frame. If selective multicasting is used and a multicast forwarding entry is found from the database, the frame is forwarded to the set of bridge ports for that multicast address group by calling the `br_multicast_forward` function. If no entry is found or selective multicasting is disabled, the frame will be handled as a broadcast Ethernet frame and forwarded by the `br_flood_forward` function.

In cases where the destination MAC address of the incoming frame is multi- or broadcast, the bridge device is set to receive all traffic, or the address is matches one of the local interfaces, a clone of the frame is also delivered upwards in the local network stack by calling the `br_pass_frame_up` function. The function updates the bridge device statistics, and passes the incoming frame up the network stack by calling the device independent `netif_receive_skb` function, ending the bridge specific processing for the frame.

The forwarding logic of the Linux bridge module is implemented in three functions: `br_forward`, `br_multicast_forward`, and `br_flood_forward`, to forward unicast, multicast, and broadcast or unknown unicast destination Ethernet frames, respectively. The simplest of the three, the `br_forward` function checks whether the destination bridge interface is in forwarding state, and then either forwards the incoming frame as is, clones the frame and forwards the cloned copy instead by calling the `deliver_clone` function, or doing nothing if the bridge interface is blocked. The `br_multicast_forward` function performs selective forwarding of the incoming Ethernet frame out of all of the bridge interfaces that have registered multicast members for the destination multicast address in the Ethernet frame, or on interfaces that have multicast routers behind them. The `br_flood_forward` function iterates over all of the interfaces in the bridge, and delivers a clone of the frame through all of them except the originating interface. Finally, all three types of forwarding functions end up calling the `__br_forward` function that actually transfers the frame to the lower layers by calling the `dev_queue_xmit` function of the interface.

3. TECHNICAL DETAILS

The Linux bridge module has two specific components that are explored in detail in this section. First, the functionality of the forwarding database is described in detail. Secondly, an overview of the IGMP snooping and selective multicasting subsystem of the Linux bridge is given, concentrating on the functional parts of the design.

3.1 Forwarding database

The forwarding database is responsible for storing the location information of hosts in the LAN. Figure 4 shows the indexing mechanism for the forwarding table, and the structure of the forwarding database array. Internally, the forwarding database is an array of 256 elements, where each element is a singly linked list holding the forwarding table entries for the hash value. The hash value for all destination MAC addresses is calculated by the `br_hash_mac` function.

The hashing process begins by extracting the last four bytes of the MAC address, creating a 32 bit identifier. The last four bytes are chosen because of the address organization in MAC addresses. Each 48 bit address consists of two parts. The first 24 bits specify an Organizationally Unique Identifier (OUI) that is assigned to the organization that issued the address. The last 24 bits specify an identifier that is unique within the OUI. The fragment of the MAC value

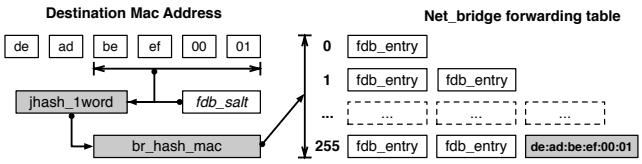

Figure 4: Linux bridge forwarding table indexing

used by the bridge contains a single byte of the OUI and all three bytes of the OUI specific identifier. This guarantees a sufficiently unique identifier, while still allowing efficient hashing algorithms to be used.

The MAC address fragment, along with a randomly generated `fdb_salt` value is passed to a generic single word hashing function in the Linux kernel, called `jhash_1word`. The resulting 32 bit hash value is then bounded to the maximum index in the hash array (i.e., 255) to avoid overflowing. The forwarding table entry for the destination MAC address is found by iterating over the linked list of the hash array element, pointed by the truncated hash value.

Unused entries in the forwarding table are cleaned up periodically by the `br_cleanup` function, that is invoked by the garbage collection timer. The cleanup operation iterates over all the forwarding database entries and releases expired entries back to the forwarding table entry cache. During iteration, the function also keeps track of the next invocation time of the cleanup operation. This is done by keeping track of the next expiration event after the cleanup invocation, based on the expiration times of the forwarding table entries that are still active during the cleanup operation.

3.2 IGMP Snooping

The IGMP snooping features of the Linux kernel bridge module allows the bridge to keep track of registered multicast entities in the local area network. The multicast group information is used to selectively forward incoming multicast Ethernet frames on bridge ports, instead of treating multicast traffic the same way as broadcast traffic. While IGMP is a network layer protocol, the IPv4 multicast addresses directly map to Ethernet addresses on the link layer. Concretely, the mapping allows local area networks to forward IPv4 multicast traffic only on links that contain hosts that use it. This can have a significant effect in the traffic characteristics of the local area network, if multicast streaming services, such as IPTV are used by several hosts.

IGMP snooping functionality consists of two parts in the Linux kernel: First, multicast group information is managed by receiving IGMP messages from end hosts and multicast capable routers on bridge ports. Next, based on the multicast group information, the forwarding decision of the bridge module selectively forwards received multicast frames on the ports that have reported a member on the multicast group address in the Ethernet frame destination address field. This paper discusses the latter part of the operation by going over the details of the multicast forwarding database, and the multicast forwarding database lookup.

Figure 5 presents an overview of the multicast forwarding

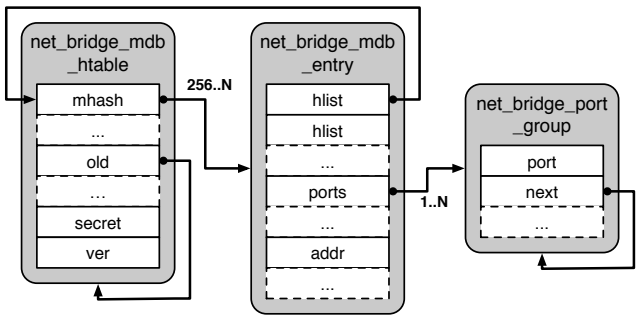

Figure 5: Linux bridge multicast forwarding database structure

database structure, and the relationships between the main data structures. The multicast forwarding database is contained in the `net_bridge_mdb_htable` data structure. The field `mhash` points to a hash array of linked list objects, similar to the normal forwarding database. The significant difference between the normal forwarding database and the multicast forwarding database is that the hash table is dynamically resized, based on the number of multicast groups registered by the operating system, either from local or remote sources. To support the efficient resizing of the database, a special field `old` is included in the data structure. This field holds the previous version of the multicast forwarding database. The previous version is temporarily stored because the rehashing operation of the multicast forwarding database is done in parallel with read access to the previous database. This way, the rehashing operation does not require exclusive access to the multicast forwarding database, and the performance of the multicast forwarding operation does not significantly degrade. After the rehash operation is complete, the old database is removed. Finally, the data structure also contains the field `secret`, that holds a randomly generated number used by the multicast group address hashing to generate a hash value for the group.

Each multicast group is contained in a `net_bridge_mdb_entry` data structure. The data structure begins with a two element array `hlist`. These two elements correspond to the position of the multicast group entry in the two different versions of the multicast forwarding database. The current version of the multicast forwarding table is defined by the `net_bridge_mdb_htable->ver` field, that will be either 0 or 1. The `ports` field contains a pointer to a `net_bridge_port_group` data structure that contains information about a bridge port that is a part of this multicast group. Finally, the `addr` field contains the address of the multicast group.

The third primary data structure for the multicast forwarding system is the `net_bridge_port_group`. The data structure holds a pointer to the bridge *port*, and a pointer to the next `net_bridge_port_group` object for a given `net_bridge_mdb_entry` object. The data structure also contains the multicast group address and various timers related to the bookkeeping of the multicast group information.

The multicast forwarding database lookup is similar to the forwarding table lookup. Figure 6 presents an overview of the operation. The hashing function takes two separate

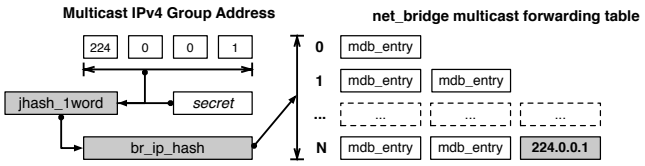

Figure 6: Linux bridge multicast forwarding database indexing

values and passes them to a generic hashing function in the Linux kernel (e.g., `jhash_1word`), similar to the MAC address hashing operation. For IPv4, the full multicast group address and the contents of the field `net_bridge_mdb_htable->secret` field are passed to the hashing function, resulting in a hash value. IPv6 uses a different hashing function that takes the full 128-bit address as an array of 4 32-bit integers. The hash value is then bounded to the maximum index of the multicast forwarding database hash array. As with the normal forwarding table, the correct `net_bridge_mdb_entry` is found by iterating over all the elements in the linked list, pointed by the bounded hash value.

4. EXPERIMENTATION

Packet processing on generic hardware is generally seen as memory intensive work [3, 4]. The experimentation in this paper explore the processing distribution between the different components of the system during the forwarding process.

4.1 Evaluation Setup

Figure 7 presents the experiment environment. It consists of a Spirent Testcenter traffic generator, and a Linux server using kernel version 3.5.3, with a bridge acting as the Device Under Test (DUT). The Spirent Testcenter generates a full duplex stream of Ethernet frames that are forwarded by the DUT using two 1Gbps network interface ports. The Linux kernel on the server collects performance statistics during the tests using the built-in profiling framework in the kernel.

The performance framework is controlled from the user space by the *perf* tool [13]. The tool offers commands to manage the performance event data collection, and to study the results. To collect performance event data, the user defines a list of either pre-defined performance events that are mapped to CPU-specific performance events by the tool, or raw performance events that can typically be found from the reference guide of the CPU or architecture model.

To generate usable performance event data, the Spirent Testcenter was used to run the RFC 2889 [9] forwarding test with 64 byte frames to determine the maximum forwarding rate of the DUT. The forwarding test performs several forwarding runs, and determines the maximum forwarding rate of the DUT by using a binary search like algorithm to narrow the forwarding rate to within a percent of the maximum. Then, five separate tests using the maximum forwarding rate with performance event data collection were run with one and 1024 Ethernet hosts on each port. The reason the performance event data collection was done this way was to eliminate the effects of frame discarding from the results, due to receiving too many frames from the traffic generator.

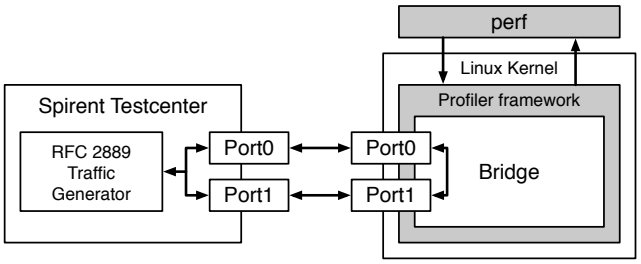

Figure 7: Experiment environment

The kernel was instrumented to collect two different performance events during the testing: used clock cycles, and cache references and misses. The cycles can be used as an estimator on the distribution of CPU processing inside the kernel. Cache references and cache misses can be used to estimate the workload of the memory subsystem in two ways. Each cache reference and miss can be likened to an operation in the CPU, that requires the program to access the main memory of the system. Cache reference occurs, when the accessed information is found in a cache, avoiding an expensive main memory access. Conversely, a cache miss happens when the information is not available in any of the caches of the CPU, and the operation requires an expensive access to the main memory of the computer.

4.2 Results

Table 1 presents the distribution of work between the different subsystems of the Linux kernel during the forwarding test with 64 byte frames. The results are given as a percent of the total number of event counters collected in the tests. The work is divided into four different subsystems: the bridge module, the network interface card driver and the network device API, the locking mechanism of the kernel for shared data structures, and the memory management.

Table 1: Performance event data distribution for RFC 2889 forwarding test

	Hosts					
	2	2048	2	2048	2	2048
Subsystem	**Cycles%**		**Cache Ref%**		**Cache Miss%**	
Interface	45.7%	40.5%	55.0%	42.2%	77.5%	77.9%
Bridge	21.0%	29.2%	11.1%	31.5%	4.2%	3.8%
Memory/IO	19.6%	17.2%	28.8%	22.0%	5.1%	5.4%
Locks	13.7%	13.2%	5.2%	4.3%	13.2%	12.9%

Almost 46% of the CPU cycles are spent in the device driver and the network device independent layer of the network stack. Next, the Linux bridging module and the memory management of the kernel are spending roughly 20% of the cycles each. Finally, the locking mechanism of the kernel is taking up the last 15% of cycles. As the number of hosts in the test increases from two to 2048, we can see that the bridge uses a larger portion of the overall cycles. The increase in used cycles is related to the organization of the hash array in the forwarding database.

The network interface and the device driver are also responsible for 55% of the cache references, and 78% of the cache misses, when there are two hosts in the LAN. We can also see a similar trend with the Linux bridging module here, as with

the cycle use. When the number of hosts increases from two to 2048, the Linux bridging module uses significantly larger portion of memory operations (and thus, caching operations) to update and query the forwarding database.

Table 2 presents the distribution of work in the Linux bridge module between the four busiest functions during the forwarding test. The results are given as a percent of the total number of event counters collected in the tests. Note that the table only holds four of the 13 different functions that participate in the DUT forwarding operation.

Table 2: Performance event distribution for RFC2889 forwarding test in the bridge module

	Hosts					
	2	2048	2	2048	2	2048
Function	**Cycles%**		**Cache Ref%**		**Cache Miss%**	
nf_iterate	19.6%	13.2%	2.3%	3.4%	12.1%	8.8%
br_fdb_update	18.2%	26.1%	42.0%	39.0%	0.1%	0.3%
br_handle_frame	13.5%	8.6%	2.7%	1.1%	3.7%	6.9%
__br_fdb_get	10.0%	23.6%	41.3%	42.9%	0.1%	0.6%

The most interesting piece of information can be seen in these results. During testing, most cycles in the Linux bridging module are not used by a bridge-specific function. The `nf_iterate` function is used by the *netfilter* module to iterate over the rules that have been specified in the system. All of the work performed by the `nf_iterate` function during the frame forwarding tests is essentially wasted, as the system had no netfilter rules defined nor does the bridging module require netfilter for any operational behavior.

We can also see from the table that most of the memory related operations are performed by the two forwarding database functions `br_fdb_update` and `__br_fdb_get`. When the number of hosts during testing is increased to 2048, the two functions also consume most of the cycles during testing. The reason for the increased processor cycle usage with increased number of hosts is explained by the architecture of the forwarding database. As mentioned in 3.1, the forwarding database consists of an array of 256 elements, where each element is a linked list. The hashing function assigns the forwarding database entry for the MAC address to one of the linked lists. Thus, the more hosts the system sees, the longer the average length of the chain for a single linked list will become. The entries in the linked lists are in arbitrary order, which requires a linear seek through the full list. This significantly increases the number of clock cycles required to find the MAC address from the linked list.

As can be seen from the table, the number of cache references stays roughly the same while the number of hosts is increased. In addition, the forwarding database in both cases fits into the system cache, as the number of misses during the forwarding database functions is insignificant. The majority of cache misses occur in the various netfilter related functions of the overall frame processing.

5. CONCLUSION

Ethernet based LANs are the building block of IP based networks, and the network application ecosystem. Local area networks are built by bridges that connect multiple Ethernet links into a single larger Ethernet cloud.

The Linux kernel contains a bridge module that can be used to create local area networks by combining network interface ports of a computer under a single bridge. While Linux bridges are not able to compete with specialized vendor hardware in performance, Linux bridging can be used in environments where performance is not the priority.

The experimentation conducted for this paper explored the performance characteristics of the Linux kernel during the bridge operation. The results show that most of the processing time is consumed by the device driver and the network interface, instead of the bridge. We can also see that modern most of the packet forwarding to occur inside the caches of the CPU. The evaluation also shows a significant increase in processing requirements in the bridge module, when the number of hosts in the LAN is significant increased.

6. REFERENCES

[1] D. Allan et al. Shortest path bridging: Efficient control of larger ethernet networks. *IEEE Communications Magazine*, 48:128–135, Oct. 2010.

[2] G. Chiruvolu, A. Ge, D. Elie-Dit-Cosaque, M. Ali, and J. Rouyer. Issues and approaches on extending Ethernet beyond LANs. *Communications Magazine, IEEE*, 42(3):80 – 86, March 2004.

[3] N. Egi et al. Towards high performance virtual routers on commodity hardware. CoNEXT '08. ACM.

[4] N. Egi et al. Forwarding path architectures for multicore software routers. PRESTO '10. ACM, 2010.

[5] W. Fenner. Internet Group Management Protocol, Version 2. RFC 2236, Internet Engineering Task Force, November 1997.

[6] A. Greenberg et al. VL2: a scalable and flexible data center network. In *SIGCOMM*. ACM, 2009.

[7] C. Guo et al. Dcell: a scalable and fault-tolerant network structure for data centers. In *SIGCOMM*, pages 75–86. ACM, 2008.

[8] C. Guo et al. BCube: a high performance, server-centric network architecture for modular data centers. In *SIGCOMM*. ACM, 2009.

[9] R. Mandeville and J. Perser. Benchmarking Methodology for LAN Switches. RFC 2889, Internet Engineering Task Force, August 2000.

[10] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. Mogul. SPAIN: COTS data-center ethernet for multipathing over arbitrary topologies. In *NSDI*. USENIX, 2010.

[11] R. Niranjan Mysore et al. PortLand: a scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*, pages 39–50. ACM, 2009.

[12] Opensolaris rbridge (IETF TRILL) support. `http://hub.opensolaris.org/bin/view/Project+rbridges/`.

[13] perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org`.

[14] R. J. Perlman. Rbridges: Transparent routing. In *INFOCOM*, pages 1211–1218, 2004.

[15] Media Access Control (MAC) Bridges. Standard 802.1D, IEEE, 2004.

[16] Virtual Bridged Local Area Networks. Standard 802.1Q-2005, IEEE Computer Society, 2005.