Optimizing the FreeBSD IP and TCP Stack

André Oppermann
andre@FreeBSD.org

The FreeBSD Project
Sponsored by TCP/IP Optimization Fundraise 2005

Abstract

FreeBSD has gained fine grained locking in the network stack throughout the 5.x-RELEASE series cumulating in FreeBSD 6.0-RELEASE [1][2]. Hardware architecture and performance characteristics have evolved significantly since various BSD networking subsystems have been designed and implemented. This paper gives a detailed look into the implementation and design changes in FreeBSD 7-CURRENT to extract the maximum network performance from the underlying hardware.

General

Performance is a very flexible term describing many different aspects on many different layers of a system. Performance can be measured and presented in many different ways. Some are meaningful and realistic, some are nice but unimportant in the big picture. Without focusing on the right metrics and overall goals one may spend a lot of time and effort optimizing one little aspect of a system without much helping the overall cause. It is just as important to find a good trade-off between short-cut optimizations and sound design with future proof system architecture. Often it is more beneficial in the mid to long run to properly analyze the big picture and then to decide how to re-implement a particular part of the system. Many times some micro optimizations should not be done to avoid architecture and layering violations preventing future changes or portability to other – newer – platforms. Not everything that is true today will be true in a few years. The same holds for optimizations that were made years ago – not every computer is a VAX. However sound system and sub-system design is very likely to be still relevant and appropriate in years to come [3].

Defining Performance

Two primary performance measures exist: Throughput and Transaction performance. Throughput is about how much of raw work can be processed in a given time interval. Transaction performance is about how many times an action can be performed in a given time interval. Most of the time these two are directly related to each other. When I'm able to perform more actions in the same amount of time I get more work done if the work size stays the same. Or the other way around when I'm able to increase the size of each work item while performing the same amount of actions the overall performance increases. What is important to note is that both of these properties have different limitations and scaling behavior. Many workloads are limited by either throughput or transactions, not both. If we can find a way to increase the limiting factor by

taking a different programmatic approach we have succeeded. It is the goal and purpose of an operating system to provide its services to the application as fast and as close to the raw underlying hardware performance as possible.

This paper examines the issues and (proposed) solutions in FreeBSD 7-CURRENT according to the layers of the OSI layered network model [4] from bottom to the top.

Physical Layer

On this layer the operating system guys have very little influence. What we can predict is that the hardware engineers are pushing the envelope on how many bits per second we can transfer over various metallic copper pairs, optical fibers and over the air. In the copper and optically wired world we are approaching 10 gigabits per second speeds as a commodity in a single stream. 40 gigabits per second is available in some high end routers already but not yet on machines FreeBSD is capable of running on [5]. However it is only a matter of time until it will arrive there too.

Data Link Layer

On the data link the world has pretty much consolidated itself to Ethernet everywhere [6]. Ethernet is packet format (called frame on this layer) with a frame payload size from 64 bytes to 1500 bytes [7]. From Gigabit Ethernet on larger frame sizes – called jumbo frames – of up to 16 kilobytes have been specified [8].

	PPS @64	PPS @1500	Payload @1500
10Mbps	14'881	813	9752926
100Mbps	148'810	8'127	97'529'259
1Gbps	1'488'095	81'274	975'292'588
10Gbps	14'880'952	812744	9752925878
40Gbps	59'523'810	3'250'975	39'011'703'511

Figure 1 *Maximal packet and payload rate at various Ethernet speeds (Source: Author)*

When a frame is received by a network interface it has to be transferred into the main memory of the system. Only there the CPU may access and further process it. This process is called DMA (direct memory access) where the network adapter writes the received frame into a predetermined location in the system memory. The first bottleneck encountered is the bus between network adapter and system memory. Network adapters almost universally use the PCI bus in its various incarnations [9]. Some manufacturers like Intel have created a direct bus between the network chip and the northbridge (memory controller) of the system to avoid PCI bus overheads. The PCI and PCI-X bus are

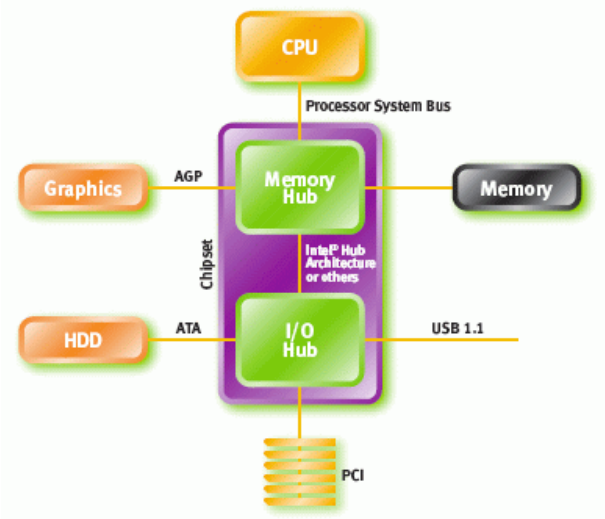

Figure 2 *Today's PC architecture (Source: PCISIG)*

designed for concurrent access by multiple devices and splits large data transfers into smaller chunks of a lower number of bytes each. This limits the maximum practically reachable throughput. For 100 Mbit Ethernet the 32 bit wide and 33MHz fast original PCI bus is sufficient even in the presence of other data

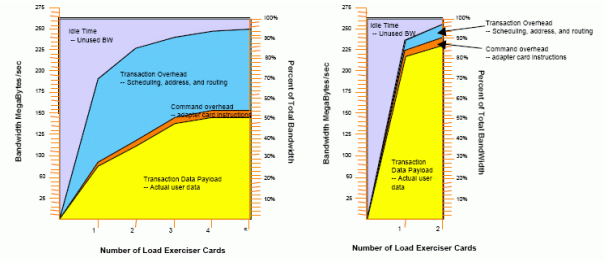

Figure 3 *PCI vs. PCI-X overhead and useable bandwidth (Source: PCISIG)*

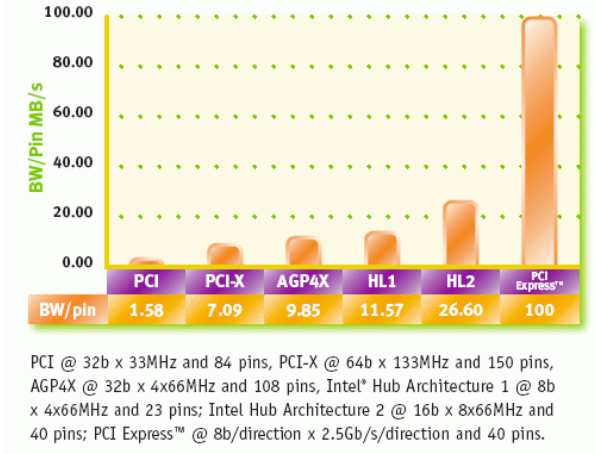

Figure 4 *PCI Bandwidth comparison (Source: PCISIG)*

transfers (hard disk access, etc). To achieve full throughput for Gigabit ethernet the extended 64 bit wide and 66-133MHz fast PCI-X is necessary. For 10 Gigabit ethernet either 64 bit and 133MHz fast PCI-X or the new point-to-point packet oriented PCI-Express bus is required. PCI-Express has many advantages compared to PCI and PCI-X. All devices have an exclusive direct connection to either the northbridge or a high speed switching fabric. The electrical connections are high speed LVDS links [10]. A number of these links can be bundled together. On the protocol level the PCIe bus works in a packet oriented mode and can transfer large chunks of data consecutively. These properties mesh ideally with the packet oriented nature of ethernet network connections.

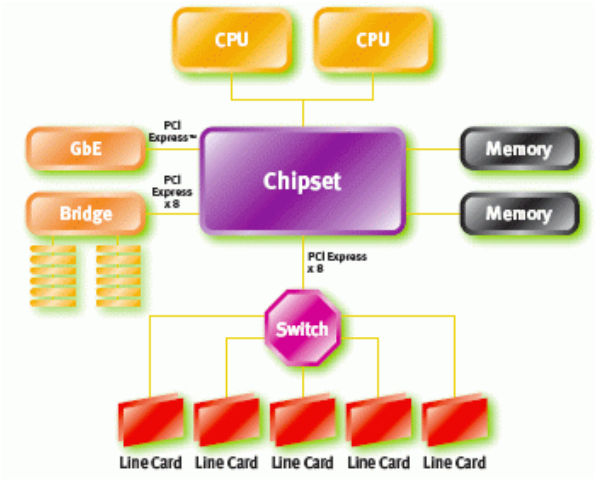

Figure 5 *PCI-Express based networking communication system (Source: PCISIG)*

Figure 6 *PCI-Express LVDS link (Source: PCISIG)*

From the operating system point of view network adapters have a number of good and bad properties. On the good side they support full wire speed on the ethernet and on the system interface side. Advanced features include IP, TCP and UDP checksum offloading and interrupt moderation. Unfortunately many ethernet chips have a number of bugs and restrictions which limit them often in very serious ways. Common problems are DMA alignment restrictions where the beginning of a frame must fall on some specific address granularity which is coarser than the general CPU platform alignment. In these cases the network driver has to copy the frame – by using the CPU – into another place to guarantee proper alignment. Obviously this doubles the workload per frame and must be avoided for high network

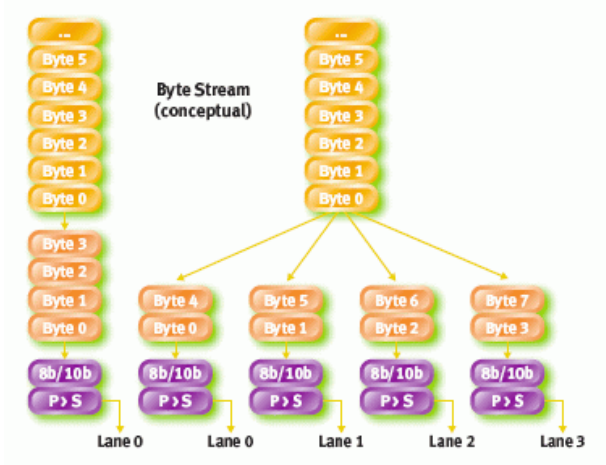

Figure 7 *PCI-Express parallel lane multiplexing (Source: PCISIG)*

performance. Very often network chips have other implementation bugs that make the advanced features unuseable. IP, TCP and UDP checksumming is often not correctly implemented and gives wrong results for certain bit pattern in frames. Here the only option is to

disable that feature and continue to calculate checksums with the CPU. Recently another advances feature called TCP segmentation offloading has been implemented in a couple of high end network cards. This feature is generally only useful when the machine is a sender of bulk TCP transfers. The net performance benefit of this offloading remain dubious and many of the implementations are again plagued by subtle bugs rendering the feature worthless. More on this in the transport layer chapter.

Cache Prefetch

Once the packet is in system memory the CPU has to start looking at the headers to determine what kind of packet it is and what to do with it. Modern CPU's run internally at many times the speed of their external system memory and employ fast cache memories close to the CPU to mitigate the effect of slow main memory accesses. Since the packet came freshly from the network it doesn't have a chance to be in the cache memories. On the first access to the packet header the CPU has to access slow system memory and to wait for a cache line to be transferred. This time is entirely lost time and occurs for every packet that enters the system at least once. Depending on the cache line size it may occur a second time when further TCP and UDP header are examined. Aware of this situation CPU designers have introduced a feature called cache prefetching whereby the programmer signals the CPU that it will access

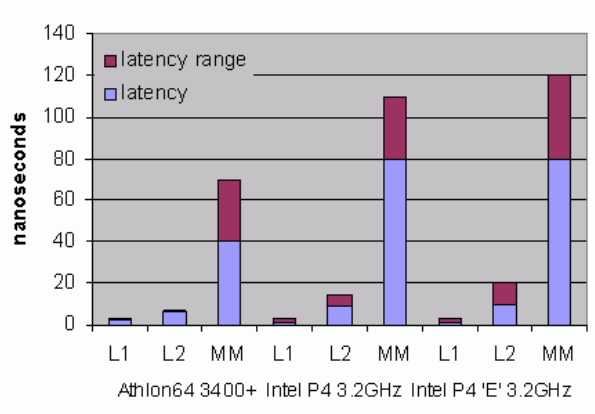

Figure 8 *Main memory access latency on cache miss (Source: Techreport)*

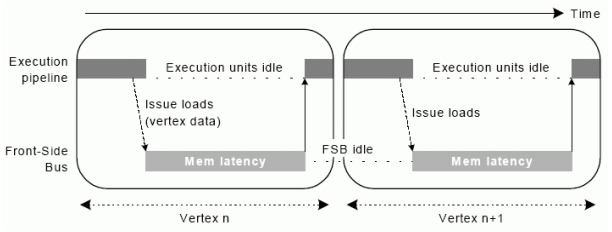

Figure 9 *Execution stall due to cache miss and memory access latency (Source: Intel)*

a certain memory region very soon [11][12][13]. The CPU can then pre-load one or more cache line sizes worth of data into the fast caches before they are actually accessed and thus avoids a full execution stall waiting for system memory. FreeBSD 7-CURRENT is gaining generic kernel infrastructure to support these cache prefetch instructions in a first implementation for Intel's Pentium 3, Pentium 4, Pentium M and AMD's Athlon, Athlon64 and Opteron series of CPUs. This prefetch command is then executed on the packet headers the very moment the network stack becomes aware of the new packet avoiding a cache stall.

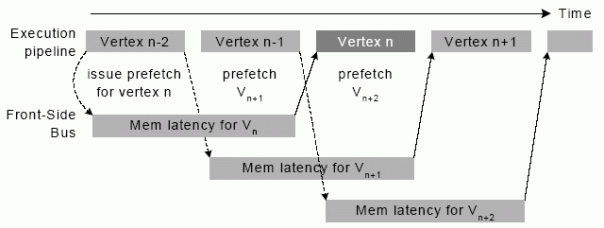

Figure 10 *Memory access latency and execution stall masked by prefetch (Source: Intel)*

Network Layer

With the packet in system memory the network hardware doesn't play a role anymore and we are squarely in the domain of the operating system network stack.

Integrity Checks

At first the network code does basic IP header integrity checks rejecting all packets with inconsistent information. Packet rejections at this stage are very seldom because a broken packet normally gets rejected by the first hop it makes.

Firewalling

After the basic integrity checks the packet is run through the firewall code. FreeBSD has three different firewall packages – ipfw2, pf (from OpenBSD) and ipf. All three firewall packages, when enabled, insert themself into the packet flow through a generic mechanism called PFIL hooks. PFIL hooks can accommodate an arbitrary number of consecutively run packet filters. To protect the integrity of the TAILQ implemented packet filter list a global, multi reader / single writer lock is asserted. Locks are expensive operations on SMP systems because they perform a synchronous write to a certain memory location. Any change to that location causes that cached information on all other CPU's to be declared invalid. Any new lock access has to obtain this memory location from slow system memory again causing an execution stall. In FreeBSD 7-CURRENT this per packet overhead is getting replaced with a lock-free but SMP safe function pointer list featuring atomic writes for changes making read locks unnecessary. Currently two implementations are proposed and performance tests will determine which one will be used.

Local or non-Local

The next step in packet processing is to determine whether the packet is for this host or if it has to be forwarded (routed) to some other system. The determination is made by comparing the destination address of the packet to all IP addresses configured on the system. If one of them matches, the packet is scheduled for further local processing. If not – and the system is acting as a router – it is scheduled for a routing table lookup. Otherwise it gets dropped and an ICMP error message is sent back to the packets source IP address. The destination address comparison used to loop through all interfaces structures and all configured IP address on them. This became very inefficient for larger number of interfaces and addresses. Already in FreeBSD 4 a hash table with all local IP addresses has been introduced for faster address compares. The probability that the hash table is permanently in cache memory is very high. Nonetheless this issue has to be further examined in detail for FreeBSD 7-CURRENT and further optimizations may be made.

Packets for a local IP addresses are discussed in the next chapter.

Routing Packets

For packets that have to be forwarded to another system, a routing table lookup on the destination address has to be performed. The routing table contains a list of all known networks and a corresponding next hop address to reach them. This table is managed by a routing daemon application implementing a routing protocol like OSPF or BGP. At the core of the Internet is a zone called DFZ (default free zone) where all globally reachable IPv4 networks are listed. At the time of writing the DFZ has a size of 173,000 network entries [14][15]. IP routing

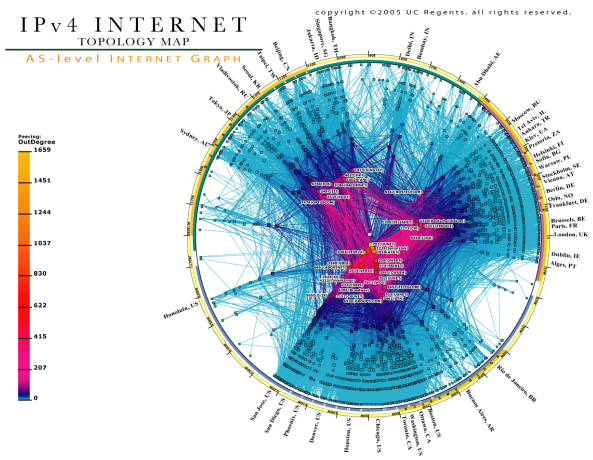

Figure 11 *IPv4 Internet DFZ topology map (Source: CAIDA)*

uses a system of longest prefix match called CIDR (classless inter-domain routing) [16][17][18]. Each network is represented by a prefix and a mask expressed in consecutive enabled bits showing the number of relevant bits for a routing decision. Such a prefix looks like this: 62.48.0.0/19 whereas 62.48.0.0 is the base aligned network address and /19 is how many

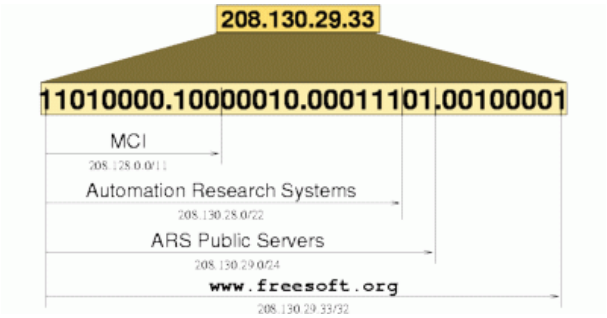

Figure 12 *CIDR Address (Source: Wikipedia)*

bits from the MSB are to be examined. In this case 19 bits making a netmask of 255.255.224.0. This entry spans 8,192 consecutive IP addresses from 62.48.0.0 to 62.48.31.255. Any prefix may have a more specific prefix covering only a part of its range or it may be a more specific prefix to an even larger, less specific one. The rule is that the most specific entry in the routing table for a destination address must win.

The CIDR system makes a routing table lookup more complicated as not only the prefix has to be looked up but also the mask has to be compared for a match. So a simple hash table approach is ruled out. Instead a trie (re***trie***val algorithm) with mask support must be used. The authors of the BSD IP stack opted for a generic and well understood PATRICIA (Practical Algorithm to Retrieve Information Coded in Alphanumeric) trie algorithm [19][20][21]. The advantage of the PATRICIA trie is its depth compression where it may skip a number of bits in depth when there is not branch in them. Thus it is able to keep the number of internal nodes very low and doesn't waste space for unnecessary ones. When a lookup is done on this tree it travels along the prefix bits as deep as possible into the tree and then compares the mask and checks if it covers the destination IP address of the packet. If not, it has to do backtracking whereas it goes one step back and compares again. This may happen until the root node of the tree is reached again and it is determined that no suitable route for this packet exists. If a match is found along the way the next hop IP address and the egress interface are looked up and the packet is forwarded to it.

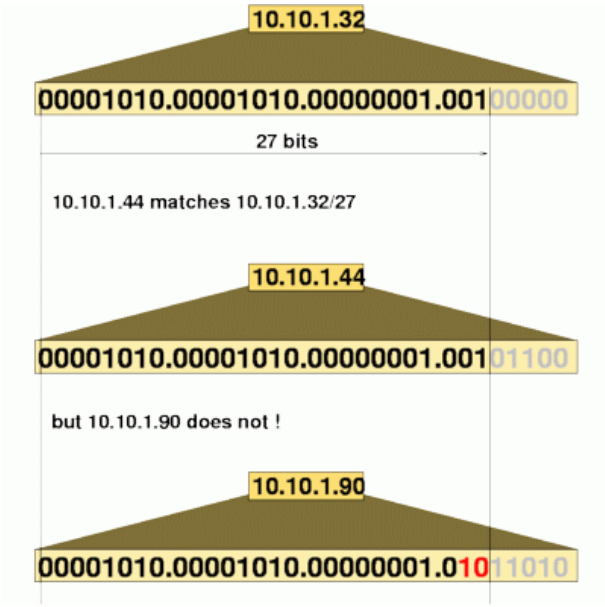

Figure 13 *IP Address Match to CIDR Prefix (Source: Wikipedia)*

With an entry count of 173,000 and backtracking the PATRICIA trie gets very inefficient on modern CPU's and SMP. For a lookup the entire tree has to be locked, plus when a matching entry was found it has to be locked too to increment its reference count when its pointer gets passed on to IP output function processing. On top of it the size of a routing entry is very large and doesn't fit into a single cache line. For a full DFZ view the BSD routing table consumes almost 50MBytes of kernel memory. It is obvious that this doesn't fit into the CPU caches and execution stalls due to slow system memory accesses happen multiple times per lookup. The larger the table gets the worse the already steep performance penalty. The worst case is a stall for every bit, 32 for IPv4.

The research literature suggest a number of different trie approaches for the longest prefix match problem [22]. A novel algorithm called LC-Trie has achieved a certain notoriety for extreme space efficiency [23][24]. It is able to represent the entire DFZ table in approximately only 3MBytes of memory on 32bit architectures. It does this by path, mask and level compression bundled with heavy pre-computation of the entire table. This algorithm is very efficient and lends itself pretty well to CPU caching. However

because it jumps around in the table it suffers from a number of execution stalls too. Nonetheless it is an order of magnitude faster than the traditional BSD trie but with one major drawback. For every change in the routing table the entire LC tree has to be re-computed, although some optimization in this area has been done [25]. This rules it out for use in an Internet environment where the constant ebb and flow of prefixes is high [26][27].

FreeBSD 7-CURRENT will implement a different but very simple, yet very efficient routing table algorithm. First it shadows the normal BSD tree and will be used only by FreeBSD's IP fast forwarding path which does direct processing to completion. Later it may become the main IPv4 routing table for the normal IP input path too. The new algorithm exploits all the positive features of modern CPU's, very fast integer computations and high memory bandwidth, while avoiding the negative cache miss execution stalls. It is very simple and it may be non-intuitive to many people accustomed to common wisdom's in computing. The algorithm splits the 32 bit IPv4 addresses into four 8 bit strides in which it has a very dense linear array containing the stride part of the prefix and its mask. It has to do at most four lookup's into four strides. The key to efficiency

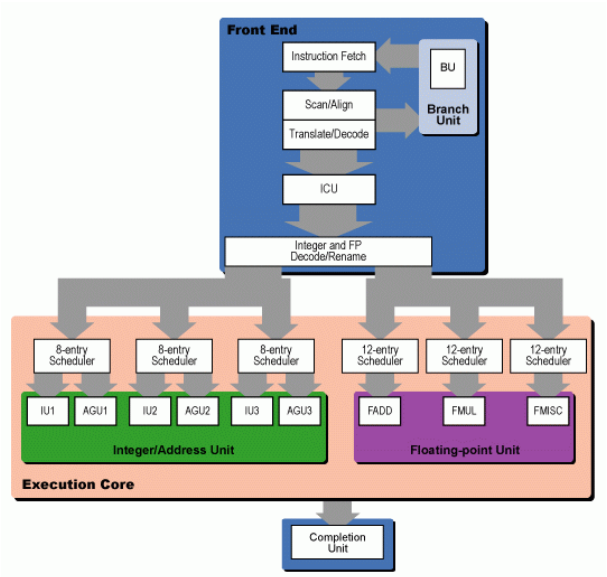

Figure 14 *AMD Athlon64/Opteron architecture (Source: Ars Technica)*

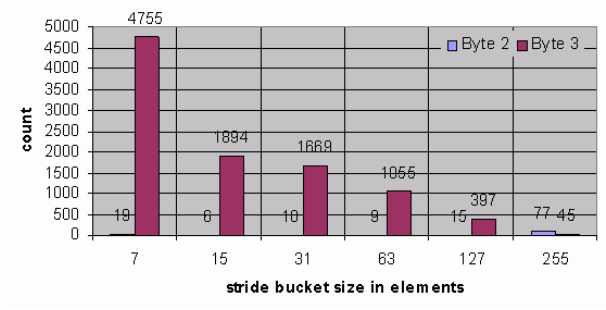

Figure 15 *Stride bucket size distribution (Source: Author)*

is cache prefetching, high memory bandwidth and fast computations. For a lookup it prefetches the first stride and linearly steps through all array entries at the level computing the match for each of them. On modern CPUs this is extremely fast as it can run in parallel in the multiple integer execution cores and all data is in the fast caches [28][29][30]. When a true match is found it is stored in a local variable. When a more specific stride match is found it prefetches that entire stride and does the same computation again for this level. Once no further strides are found the most specific match is used to forward the packet. If no match was ever found it is clear there is no routing table entry and the packet gets rejected. No backtracking has to occur. At most four, one for each stride and masked by the prefetch, execution stalls can happen. The footprint of each entry is very small and the entire table fits into approximately the same amount of space as the LC tree. It has a few important advantages however. It doesn't need any locking for lookup. Lookups can happen in parallel on any number of CPUs and it allows for very easy and efficient table updates. For writes to the tables a write lock is required to serialize all changes and prevent multiple CPUs from updating entries at the same time. While a change is made lookups can still continue. All changes are done with atomic writes in the correct order. This gives a coherent view of the table at any given point in time. Many changes – next hop, invalidation of prefix, addition of a prefix when there is space left in a stride bucket – are done with just one atomic operation. All other changes prepare a new, modified stride bucket and then swap the parents

stride pointer to it. The orphaned stride bucket gets garbage collected after a few milliseconds to guarantee that any readers have left it by then. This routing table design has been inspired by the rationale behind [31].

Transport Layer

Protocol Control Block Lookup

Packets for a local IP addresses get delivered to the socket selection of their respective protocol type – commonly TCP or UDP. The protocol specific headers are checked first for integrity and then it gets determined if a matching socket exists. If not the packet gets dropped and an ICMP error message is sent back. For TCP packets, now called segments, the socket lookup is complicated. The host may have a number of active TCP connections and a number of listening sockets. To make a determination where to deliver the packet a hash table is employed again. Before the hash table lookup can be made the entire TCP control block list including the hash table has to be locked to prevent modifications while the current segment is processed. The global TCP lock stretches over the entire time the segment is worked on. Obviously this locks out any concurrent TCP segment processing on SMP as only one CPU may handle a segment at any give point in time. On one hand this is bad because it limits parallelism but on the other hand it maintains serialization for TCP segments and avoids spurious out of order arrivals due to internal locking races between CPUs handling different segments for the same session. How to approach this problem in FreeBSD 7-CURRENT is still debated. One proposed solution is a trie approach similar the new routing table coupled with a lockless queue in each TCP control block. When a CPU is processing one segment and has locked the TCPCB while another CPU has already received the next segment it simply gets attached to the lockless queue for that socket. The other CPU then doesn't has to spin on the TCPCB lock and wait for it to get unlocked. The first CPU already has the entire TCPCB structure and segment processing code in the cache and before it exits the lock it checks the queue for further segments. Some safeguards have to be employed to prevent the first CPU from looping for too long in the same TCPCB. It may have to give up further processing after a number of segments to avoid lifelock. The final approach for FreeBSD 7-CURRENT is still under discussion in the FreeBSD developer community and extensive performance evaluations will be done before settling to one implementation.

TCP Reassembly

TCP guarantees a reliable, in-sequence data transport. To transport data over an IP network it chops up the data stream into segments and puts them into IP packets. The network does its best effort to deliver all these packets. However occasionally it happens that packets get lost due to overloaded links or other trouble. Sometimes packets even get reordered and a packet that was sent later may arrive before an earlier one. TCP has to deal with all these problems and it must shield the application from them by handling and resolving the errors internally. In the packet loss case only a few packets may be lost and everything after it may have arrived intact. TCP must not present this data to the application until the missing segments are recovered. It asks the sender to retransmit the missing segments using either duplicate-ACK's or SACK (selective acknowledges) [32]. In the meantime it holds on to the already received segments in the TCP reassembly queue to speed up transmission recovery and to avoid re-sending the perfectly

ms / Mbps	**10**	**100**	**1000**
1	1	12	122
10	12	122	1'221
100	122	1'221	12'207
200	244	2'441	24'414
300	366	3'662	36'621

Figure 16 *Bandwidth * delay product in kbytes at various RTT and speeds, 300ms is Europe - Japan (Source: Author)*

received later segments. The same applies for the reordering case where usually only a small number of packets is held onto until the missing segment arrives. With today's network speeds and long distances the importance of an efficient TCP reassembly queue becomes evident as the bandwidth-delay product becomes ever larger. A TCP socket may have to hold to as many data in the reassembly queue as the socket buffer limit provides. Generally the socket buffers over-commit memory – they don't have enough physical memory to fulfill all obligations simultaneously – they may have on all sockets together. In addition all network data arrives in mbufs and mbuf clusters (2kbytes in size), no matter how much actual payload is within such a buffer. The current FreeBSD TCP reassembly code is still mostly the same as in 4.4BSD Net/2. It simply creates a linked list of all received segments and holds on to every mbuf it got data in. Obviously this is no longer efficient with large socket buffers and provides some attack vectors as well as for memory exhaustion by deliberately sending many small packets while forgetting the first one. All the memory and mbufs are then tied up in the reassembly queue and not available for legitimate data. Replicate this for a couple of connections and the entire server runs out of available memory. In FreeBSD 7-CURRENT the entire TCP reassembly queue gets rewritten and replaced with an adequate system. The new code coalesces all continues segments together and stores them as only one block in the segment list. This way only a few entries have to be searched in worst case if a new segments arrives. The author has provided a proof of concept for this part which was demonstrated to have significant benefits over the previous code on large buffers and a 4Gbps Myrinet link with constant packet reordering due to a firmware bug [33]. The proof of concept code is currently developed further to merge mbufs in the reassembly queue when either the previous or following mbuf has enough free space to store the data portion of the current one. This way a large part of the malicious attack scenarios is covered. Then to thwart all other attacks described in research papers only the number of missing segments (holes) has to be limited [34].

TCP segmentation offloading is a controversial topic and has been hyped a lot with the introduction of iSCSI and TOE (TCP Offload Engines). TOE do the entire TCP processing in dedicated processors on the network card [35]. The clear disadvantage of TOE is the operating system has no longer any control over the TCP session, its implementation and advanced features. FreeBSD has a very good TCP and IP stack and we most likely will not support full TCP offloading. In addition the benefits are limited even with TOE as the operation system still has to copy all data from and to the application from kernel space. TCP segmentation offloading (TSO) is more interesting and to some extent supported on most gigabit ethernet network cards. Unfortunately often bugs in edge cases or with certain bit patterns make this feature useless. Complicating the matter is the functioning of the general network stack in FreeBSD where every data stream is stored on mbuf clusters. The mbuf clusters are a little bit larger than the normal ethernet MTU of 1500 bytes. Thus we already have a direct natural fit which lessens the need and benefit of TSO. There are cases where TSO may be beneficial nonetheless. For example high speed single TCP connection transfers may receive a boost from lesser CPU processing load. Current experience with existing implementations is inconclusive and for FreeBSD 7-CURRENT we will do further research to judge the possible advantages against the complications of implementing support for TSO [36][37]. An implementation of TSO for FreeBSD's network stack is a non-trivial endeavor.

Session Layer

T/TCP Version 2

T/TCP stands for transactional TCP. This name however is misleading as it doesn't have anything to do with transactions commonly

understood from databases, file systems or other applications. Rather it tries to provide reliable transport that is faster than normal TCP for short connections found in many applications, most notably HTTP. It does this by modifying certain aspects and behaviors of TCP [38]. It was observed early on that the single largest latency block in short TCP connections comes from the three way handshake. T/TCP optimizes this by doing a three way handshake only the first time any two hosts communicate with each other. All

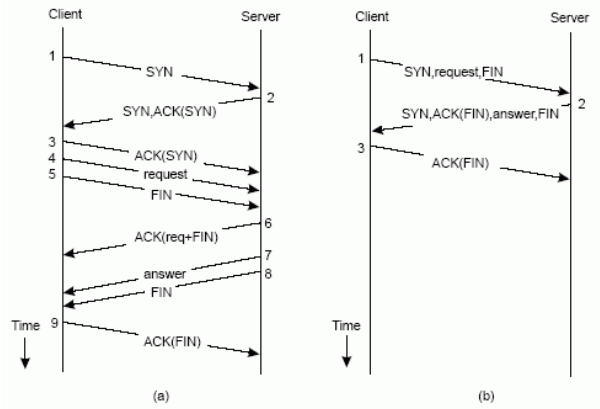

Figure 17 *Comparison between a TCP (a) and T/TCP+T/TCPv2 (b) connection setup (Source: Vrije Universiteit)*

following connections send their data/request segment directly with the first SYN packet. The receiving side then directly converts this one packet into a full socket and hands it over to the application for processing of the contained data or request instead of replying with SYN-ACK. T/TCP waits until the applications answers the request to piggyback the response with the first packet sent back. This approach is clearly very efficient and fast. Unfortunately the specification is very weak on security and the TCP part of the implementation complicated [39]. On the application side TCP connections can be opened without calling connect(2) on a socket by using send*(2) doing an implicit and automatic connect like UDP [40]. The host authentication for single packet connects uses a count of the connections between two hosts since the last three way handshake. This is very easy to spoof and to launch SYN attacks with. Thus T/TCP never gained any meaningful traction in the market as it was unfit for any use on the open Internet. The only niche it was able to establish itself to some extent is the satellite gateway market where the RTT is in the range of 500ms and everything cutting connection latency is very valuable. With partly rewriting T/TCP avoiding the weaknesses the Author tries to bring back the clear benefits it can provide. The rewrite is dubbed T/TCPv2 and will be first implemented in FreeBSD 7-CURRENT as an experimental feature. The original connection count is replaced with two 48bit random values (cookies) exchanged between the hosts. One cookie, the client cookie, is initialized by the client for all connections anew when it issues the SYN packet. This cookie is then transmitted with every segment from the client to the server and from the server to client. It adds 48bits of further true entropy to the 32bit minus window size to protect the TCP connection from any spoofing or interference attempts. This comes at very little cost with only 8 bytes overhead per segment and a single compare upon reception of a segment. It is not restricted to T/TCPv2 and can be used with any TCP session as a very light-weight alternative to TCP-MD5 [41]. The other cookie is a server cookie which is transmitted from the server to the client in the SYN-ACK response to the first connection. The first connection is required to go through the three way handshake too. This cookie value is remembered by the client and server and must be unique plus random for every client host. The client then sends it together with the SYN packet already containing data on subsequent connections to qualify for a direct socket like in original T/TCP. Unlike the previous implementation it will not wait for the application to respond but send a SYN-ACK right away to notify the client of successful reception of the packet. These two random value cookies make T/TCPv2 (and TCP with the client cookie) extremely resistant against all spoofing attacks. The only way to trick a T/TCPv2 server is by malicious and cooperating clients where the master client obtains a legitimate server cookie and then distributes it to a number of other clients which then issue spoofed SYN request under the identity of the master client.

Presentation Layer

Skipped in this paper. TCP/IP does not have a presentation layer.

Application Layer

In the application space HTTP web servers are a prime example of being very dependent on the underlying operating system and exercising the network stack to its fullest extent. A HTTP server serving static objects – web pages, images and other files – is entirely dominated by operating system overhead and efficiency [42]. A HTTP request comes in from a client as TCP session starting with a SYN packet entering the SYN cache. The network stack responds with SYN-ACK and then the client sends a request for an object. When the first part of the request is received the SYN cache expands the connection into a full socket and signals the web server the availability of a request by waking it up from listen(2). The server then accepts the request, parses it and locates the file in the file system. When it is located and the server has sufficient permissions to access it, it opens the file for reading and sends the file content to the client via the socket and closes the file again. Once the client network stack has acknowledged all packets the socket is closed on the server. The HTTP request is fulfilled. Along this path a number of potentially latency inducing steps occur. First in line are the listen(2) and accept(2) system calls dealing with all incoming connections. FreeBSD implements an extension to a socket in listen state called accept filter which may be enabled with a setsockopt(2) call. The accf_http(9) filter accepts incoming connections but waits until a full HTTP request has been received by the server until it signals the new connection to the application. Normally this would happen right after the ACK to the SYN-ACK has been received by the server. In the average case this saves a round-trip between kernel and application. All new incoming connections receive their own socket file descriptor and the application has to select(2) or poll(2) on them to check for either more request data to read or more space in the socket to send. Both calls use arrays of sockets which have to be re-initialized every time a call to these function is made. With large numbers of connections this causes a lot of overhead in processing and becomes very inefficient. FreeBSD has introduced an event driven mechanism called kqueue(2) to overcome this limitation [43]. With kqueue the application registers a kernel event on the socket file descriptor and specifies which events it is interested in. The registered event is active until it is cancelled. Whenever a specified event is triggered on any registered event, the event is added to an aggregated event queue for this application from which it can read the events one after the other. This programming model is not only highly efficient but also very convenient for server application programmers and is made easily available in a portable library called libevent [44]. Once the request has been

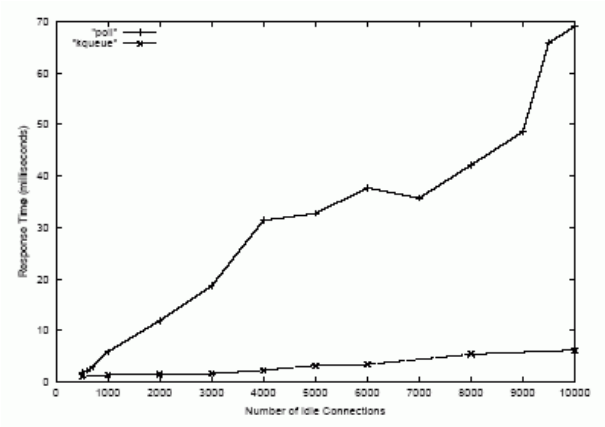

Figure 18 *Response time poll(2) vs. kqueue(2) from httperf (Source: Jonathan Lemon)*

fully received the HTTP server parses it for the requested file name and starts a lookup in the file system with stat(2). It is an often overlooked point of undesired blocking and latency when the file path and directory entry are not already in the file system or buffer cache. Reads from the disk may have to be initiated and during that time the application will block in the kernel and can't perform any other work as the stat(2) system call can't be performed in a non-blocking way. To avoid this stall in the main loop of the application it is beneficial to perform the stat(2)

outside of the main loop and distribute it among a number of pthread(2)s or pre-fork(3)ed process children. Thus the application can accept and process further incoming connections as well as quickly answer those which are already in any of the operating systems caches. After the stat(2) has determined that the file is available and the application has sufficient rights to read it, it is open(2)ed for reading. Normally the file content is read into the application and then written out on the socket again. This however causes the file content to be copied two times between the kernel and application. The sendfile(2) system call offers a direct path from the file system to the network socket. With sendfile(2) the application specifies an optional header and footer which is sent with the file, the file descriptor of the opened file, the length and the offset in the file to be sent. This approach completely eliminates any file content copies between kernel and application and it allows the kernel to coalesce header, footer and file content together into fewer, larger packets sent over the network. Here again the sendfile(2) system call may block if not all file content is in the file system or buffer cache causing the application to block until all data is fetched from the physical disk. Sendfile(2) offers an option to immediately return with an EWOULDBLOCK error message signaling the direct unavailability of the file content. The application then may use the same approach as with stat(2) and distribute it to either a pthread or pre-forked process child for further processing keeping the main loop going. FreeBSD 7-CURRENT will continue to improve the internal efficiency of the existing optimization functions and may implement further methods as outlined in [45][46].

Business Layer

The author wants to thank all sponsors of the TCP/IP Optimization Fundraise 2005 for making a lot of optimization work in the FreeBSD kernel possible. A full list of all donors and their contribution is available at [47].

References:

[1] Introduction to Multithreading and Multiprocessing in the FreeBSD SMPng Network Stack, Robert N. M. Watson, EuroBSDCon Basel, November 2005, http://www.eurobsdcon.org, http://people.freebsd.org/~rwatson/

[2] New Networking Features in FreeBSD 6.0, André Oppermann, EuroBSDCon Basel, November 2005, http://www.eurobsdcon.org, http://people.freebsd.org/~andre/

[3] The Design and Implementation of the FreeBSD Operating System, Marshall Kirk McKusick and George V. Neville-Neil, 2004, Addison-Wesley, ISBN 0-201-70245-2, http://www.aw-bc.com/catalog/academic/product/0,1144,0201702452,00.html

[4] Open Systems Interconnection Reference Model, http://en.wikipedia.org/wiki/OSI_model

[5] Cisco CRS-1 Carrier Routing System, 40Gbps (OC-768/STM-258) interface, http://www.cisco.com/en/US/products/ps5763/index.html

[6] Ethernet, General Description and History, http://en.wikipedia.org/wiki/Ethernet

[7] IEEE 802.2 Logical Link Control, 1998 Edition, http://standards.ieee.org/getieee802/download/802.2-1998.pdf

[8] IEEE 802.3 LAN/MAN CSMA/CD Access Method, http://standards.ieee.org/getieee802/802.3.html

[9] PCI-SIG PCI 2.3, PCI 3.0, PCI-X 2.0, PCI-Express 1.1 Specification, http://www.pcisig.com

[10] Low voltage differential signaling, http://en.wikipedia.org/wiki/Low_voltage_differential_signaling

[11] Software Optimization Guide for AMD64 Processors, Chapter 5, AMD Publication #25112, Revision 3.06, September 2005, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF

[12] IA-32 Intel® Architecture Optimization Reference Manual, Chapter 6, Intel Document # 248966, Revision 1.2, June 2005, ftp://download.intel.com/design/Pentium4/manuals/24896612.pdf

[13] Understanding CPU caching and performance, Jon "Hannibal" Stokes, Ars Technica, July 2002, http://arstechnica.com/articles/paedia/cpu/caching.ars

[14] CAIDA, Cooperative Association for Internet Data Analysis, Routing Analysis Group, http://www.caida.org/analysis/topology/

[15] Daily CIDR and DFZ Routing Report, Geoff Huston, July 1988 - present, http://www.cidr-report.org/

[16] RFC1518, An Architecture for IP Address Allocation with CIDR, September 1993, http://www.ietf.org/rfc/rfc1518.txt

[17] RFC1519, Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy, September 1993, http://www.ietf.org/rfc/rfc1519.txt

[18] Classless Inter-Domain Routing, Description and History, http://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing

[19] PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric, Donald R. Morrison, Journal of the ACM, Volume 15, Issue 4 (October 1968), Pages: 514 - 534, http://portal.acm.org/citation.cfm?id=321481

[20] TCP/IP Illustrated, Vol. 2, The Implementation, Gary R. Wright and W. Richard Stevens, Addison-Wesley, 1995, ISBN 0-201-63354-X, http://www.aw-bc.com/catalog/academic/product/0,1144,020163354X,00.html

[21] Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching, 3rd Edition, September 1997, ISBN 0-201-31452-5, http://www.aw-bc.com/catalog/academic/product/0,1144,0201314525,00.html

[22] On Fast Address-Lookup Algorithms, Henry Hong-Yi Tzeng, Tony Przygienda, IEEE Journal on Selected Areas in Communications, Vol. 17, No. 6, June 1999, Pages: 1067 - 1082, http://dl.comsoc.org/cocoon/comsoc/servlets/GetPublication?id=92711

[23] IP-Address Lookup Using LC-Tries, S. Nilsson and G. Karlsson, IEEE Journal on Selected Areas in Communications., Vol.17, No.6, June 1999, Pages:1083 - 1092, http://dl.comsoc.org/cocoon/comsoc/servlets/GetPublication?id=137835

[24] Modified LC-Trie Based Efficient Routing Lookup, V.C. Ravikumar, R. Mahapatra, J. C. Liu, Proceedings of the 10th IEEE MASCOTS, 2002, Page: 177, http://doi.ieeecomputersociety.org/10.1109/MASCOT.2002.1167075

[25] Enabling incremental updates to LC-trie for efficient management of IP forwarding tables, D. Pao, Yiu-Keung Li, IEEE Communications Letters, Vol. 7, No. 5, May 2003, Pages: 245 - 247, http://dl.comsoc.org/cocoon/comsoc/servlets/G

etPublication?id=1218307

[26] RIS – Routing Information Service, RIPE, 1999 - present, http://www.ripe.net/projects/ris/

[27] CAIDA, Cooperative Association for Internet Data Analysis, Routing Analysis Group, http://www.caida.org/analysis/routing/

[28] Inside AMD's Hammer: the 64-bit architecture behind the Opteron and Athlon 64, Jon “Hannibal” Stokes, Ars Technica, February 2005, http://arstechnica.com/articles/paedia/cpu/amd-hammer-1.ars

[29] The Pentium: An Architectural History of the World's Most Famous Desktop Processor (Part I), Jon “Hannibal” Stokes, Ars Technica, July 2004, http://arstechnica.com/articles/paedia/cpu/pentium-1.ars

[30] The Pentium: An Architectural History of the World's Most Famous Desktop Processor (Part II), Jon “Hannibal” Stokes, Ars Technica, July 2004, http://arstechnica.com/articles/paedia/cpu/pentium-2.ars

[31] Judy Arrays, Doug Baskins, Alan Silverstein, HP, January 2002, http://judy.sourceforge.net/doc/shop_interm.pdf

[32] RFC2018, TCP Selective Acknowledgment Options, October 1996, http://www.ietf.org/rfc/rfc2018.txt

[33] Rewritten TCP reassembly, André Oppermann, December 2004, FreeBSD-net mailing list, http://lists.freebsd.org/pipermail/freebsd-net/2004-December/005879.html

[34] Robust TCP Stream Reassembly In the Presence of Adversaries, Sarang Dharmapurikar, Vern Paxson, August 2005, http://www.icir.org/vern/papers/TcpReassembly/TcpReassembly.pdf

[35] Introduction to TCP Offload Engines, Sandhya Senapathi, Rich Hernandez, Dell Power Solution Magazine, March 2004, http://www.dell.com/downloads/global/power/1q04-her.pdf

[36] Linux and TCP offload engines, Corbet and comments, LWN.net, August 2005, http://lwn.net/Articles/148697/

[37] Response to Article on TOE, Wael Noureddine and comments, LWN.net, August 2005, http://lwn.net/Articles/149941/

[38] RFC1644, T/TCP – TCP Extensions for Transactions Functional Specification, July 1994, http://www.ietf.org/rfc/rfc1644.txt

[39] TCP/IP Illustrated, Vol. 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols, W. Richard Stevens, Addison Wesley, 1996, ISBN 0-201-63495-3, http://www.aw-bc.com/catalog/academic/product/0,1144,0201634953,00.html

[40] FreeBSD 5.4 manual page ttcp(4), http://www.freebsd.org/cgi/man.cgi?query=ttcp&apropos=0&sektion=0&manpath=FreeBSD+5.4-stable&format=html

[41] RFC2385, Protection of BGP Sessions via the TCP MD5 Signature Option, August 1998, http://www.ietf.org/rfc/rfc2385.txt

[42] Flash: An Efficient and Portable Web Server, Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel, Rice University, Proceedings of the 1999 USENIX Annual Technical Conference, http://www.usenix.org/publications/library/proceedings/usenix99/full_papers/pai/pai.pdf

[43] Kqueue: A generic and scalable event notification facility, Jonathan Lemon, FreeBSD, May 2001, http://people.freebsd.org/~jlemon/papers/kqueue.pdf

[44] libevent library, Niels Provos, www.monkey.org/~provos/libevent/

[45] Lazy Asynchronous I/O For Event-Driven Servers, Khaled Elmeleegy, Anupam Chanda, Alan L. Cox, Willy Zwaenepoel, Proceedings of the General Track, 2004 USENIX Annual Technical Conference, http://www.usenix.org/events/usenix04/tech/general/full_papers/elmeleegy/elmeleegy.pdf

[46] accept()able Strategies for Improving Web Server Performance, Tim Brecht, David Pariag, Louay Gammo, Proceedings of the General Track, 2004 USENIX Annual Technical Conference, http://www.usenix.org/events/usenix04/tech/general/full_papers/brecht/brecht.pdf

[47] FreeBSD TCP/IP Cleanup and Optimization Fundraise 2005, André Oppermann, FreeBSD kernel committer, July 2005, http://people.freebsd.org/~andre/tcpoptimization.html