# Optimizing the FreeBSD IP and TCP Stack

André Oppermann <andre@FreeBSD.org>
Sponsored by TCP/IP Optimization Fundraise 2005

EuroBSDCon 05
Basel, 27. November 2005

# About

FreeBSD has gained fine grained locking in the network stack throughout the 5.x-RELEASE series cumulating in FreeBSD 6.0-RELEASE.

Hardware architecture and performance characteristics have evolved significantly since various BSD networking subsystems have been designed and implemented.

This talk 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

Many common assumptions people made in the VAX days no longer hold true.

Many still commonly accepted rules of thumb no longer hold true.

Don’t assume anything.

Profile, don’t speculate!

In German we say: “Wer misst, misst Mist”.

There are lies, damn lies and statistics.

This Presentation and the paper is following the layers of the OSI stack starting from the physical layer.

# Performance - A Definition

What is “performance”?

Performance can be measured and presented in many different ways. Some are meaningful and realistic, some are nice but unimportant in the big picture.

Focus on the right metrics and overall goal. Don’t just focus on one little aspect which may not help a lot overall.

Find a good trade-off between short-cut optimizations.

Sound design with future proof system architecture.

Properly analyze the big picture and then to decide if and how to re-implement a particular part of the system.

# Performance - A Definition

Many times micro optimizations should not be done or are done prematurely.

Avoid architecture and layering violations preventing future changes or portability to other or newer platforms.

Not everything that is true today will continue to be true in a few years.

# Performance - A Definition

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.

It 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.

# Physical Layer

Operating system guys have very little influence.

We can predict  that the hardware engineers are pushing the envelope.

More speed 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.

It is only a matter of time until it will arrive there too.

# Link Layer - Ethernet

The data link the world has pretty much consolidated itself to Ethernet everywhere.

Ethernet is a packet format (called frame on this layer) with a frame payload size from 64 bytes to 1500 bytes.

Gigabit Ethernet and faster have larger frame sizes – called jumbo frames – of up to 16 kilobytes.

# Link Layer

Ethernet packet payload vs. PPS vs. net throughput:

| | 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 | 812'744 | 9'752'925'878 |
| **40Gbps** | 59'523'810 | 3'250'975 | 39'011'703'511 |

# Link Layer - DMA

When a frame is received by a network interface it has to be transferred into the main memory of the system.

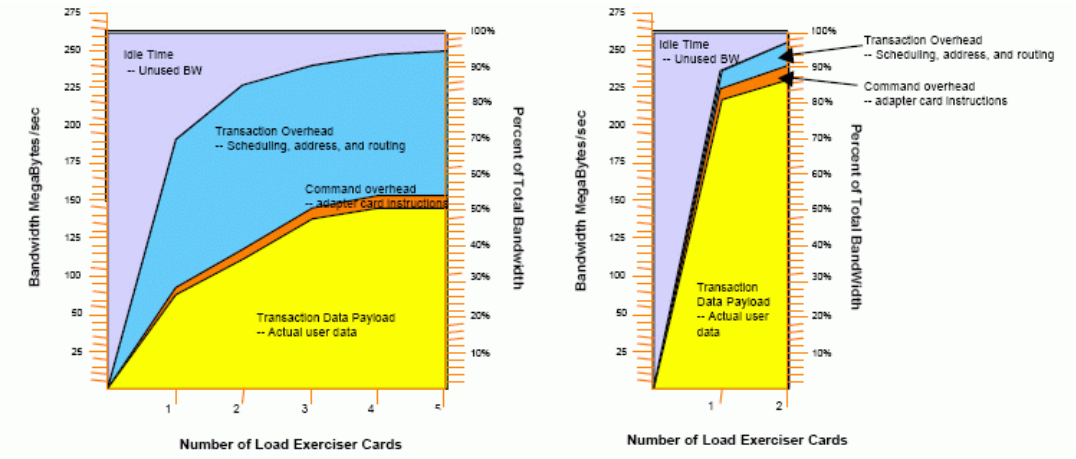
Only there the CPU may access and further process it.

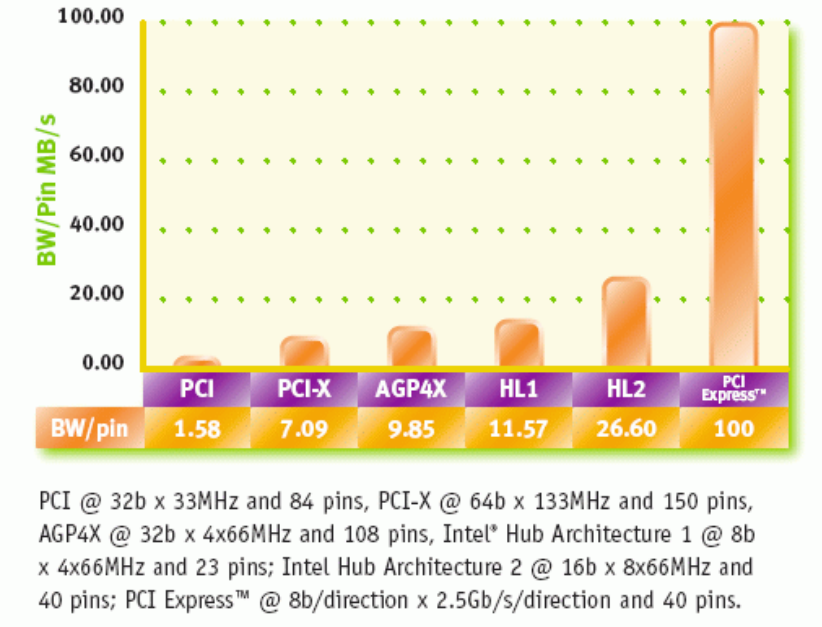
This process is called DMA (direct memory access) where the network adapter writes the received frame into a predetermined location in the system memory.

The first bottleneck encountered is the bus between network adapter and system memory.

# Link Layer - System Bus

PCI vs. PCI-X vs. PCI-Express




PCI @ 32b x 33MHz and 84 pins, PCI-X @ 64b x 133MHz and 150 pins, AGP4X @ 32b x 4x66MHz and 108 pins, Intel® Hub Architecture 1 @ 8b x 4x66MHz and 23 pins; Intel Hub Architecture 2 @ 16b x 8x66MHz and 40 pins; PCI Express™ @ 8b/direction x 2.5Gb/s/direction and 40 pins.

# Link Layer - Network adapters

Good:

Full wire speed on the ethernet and on the system interface side.

Advanced features like IP, TCP and UDP checksum offloading and interrupt mitigation.

Bad:

DMA alignment restrictions.

Bugs that make advanced features unuseable. IP, TCP and UDP checksumming is often not correctly implemented and gives wrong results for certain bit patterns.

# Link Layer - Packet Handling

After DMA the CPU has to look at packet headers.

CPU's run internally at many times the speed of their external system memory.

Packet came freshly from the network and don't have a chance to be in the cache memories.

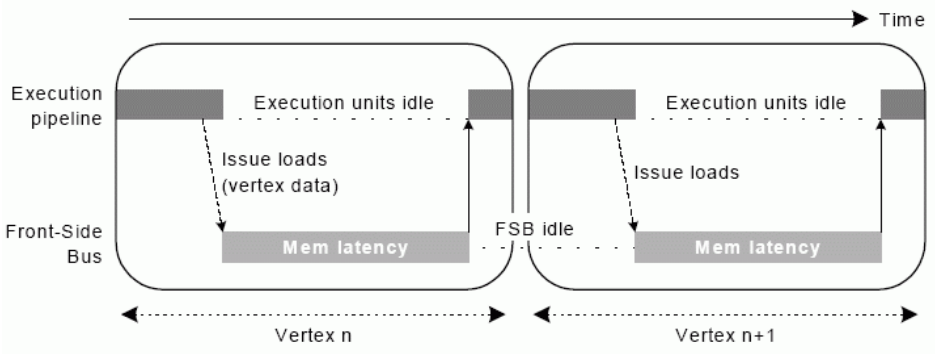
CPU has to access slow system memory and to wait for a cache line to be transferred.

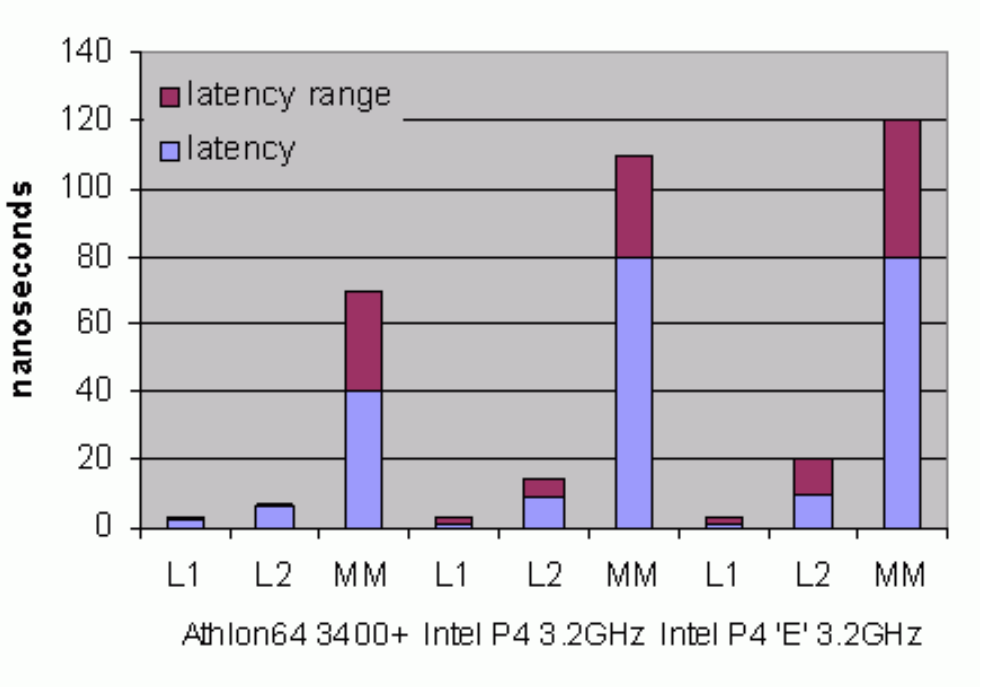
This time is entirely lost time and occurs for every packet that enters the system at least once.

Depending on the cache line size it may occur a second time when further TCP and UDP header are examined.

# Link Layer - Cache Miss

Execution stall and cache latency:



# Link Layer - Prefetching

Aware of this situation CPU designers have introduced a feature called "cache prefetching" whereby the programmer signals the CPU that it will access a certain memory region very soon.

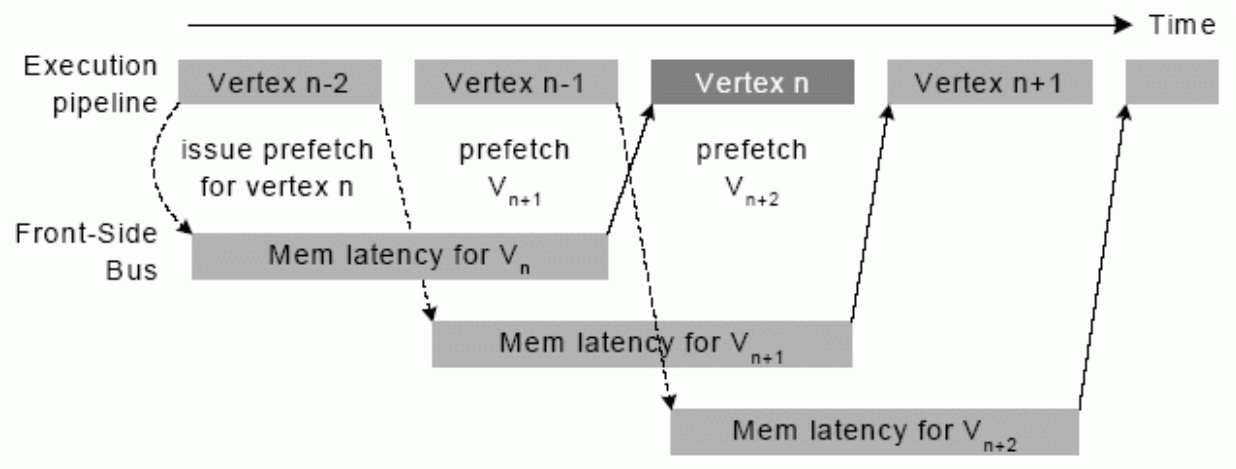
The CPU can then pre-load one or more cache line sizes worth of data into the fast caches before they are actually accessed and thus avoids a full execution stall waiting for system memory.

This prefetch command is executed on the packet headers the very moment the network stack becomes aware of the new packet avoiding a cache stall.

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.

# Link Layer - Prefetch + cache

Masked execution stall with prefetch:



# Network Layer - PFIL

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.

First the network code does basic IP header integrity checks.

Next the packet is run through the firewall code. All firewall packages insert themself into the packet flow through a generic mechanism called PFIL hooks.

FreeBSD 7-CURRENT the PFIL hook implementation is getting replaced with a lock-free but SMP safe function pointer list featuring atomic writes for changes making read locks unnecessary.

# Network Layer - Routing

The next step 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.

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.

Packets for a local IP addresses are discussed later.

# Network Layer - Routing

For packets that have to be forwarded, 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.

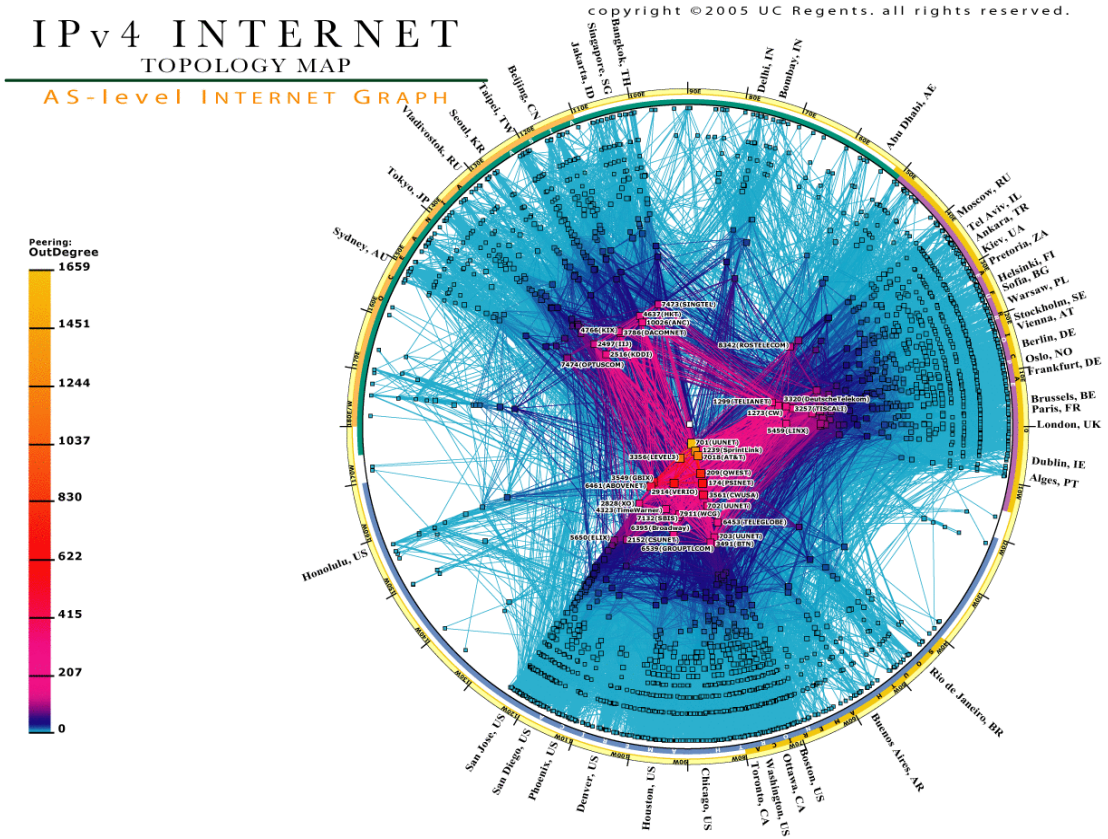
This table is managed by a routing daemon application implementing a routing protocol like OSPF or BGP.

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 175,000 network entries.

# Network Layer - DFZ

CAIDA graph:



# Network Layer - CIDR

IP routing uses a system of longest prefix match called CIDR (classless inter-domain routing).

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:

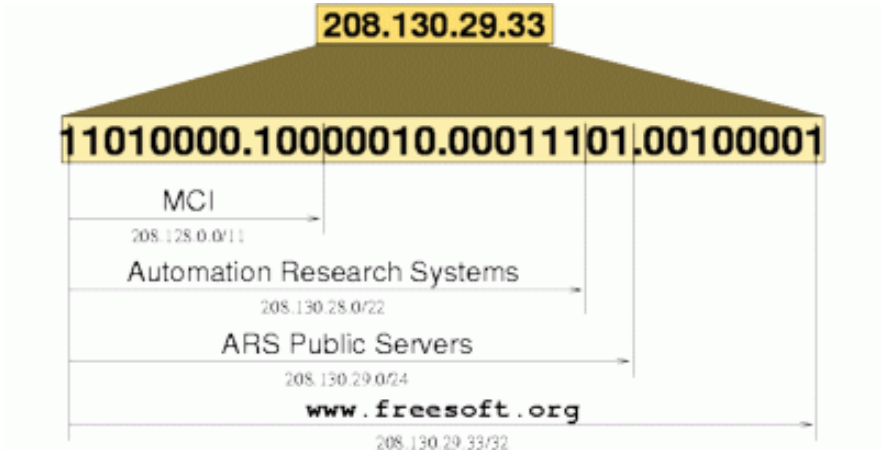
> 62.48.0.0/19 whereas 62.48.0.0 is the base aligned network address and /19 is how many bits from the MSB are to be examined.

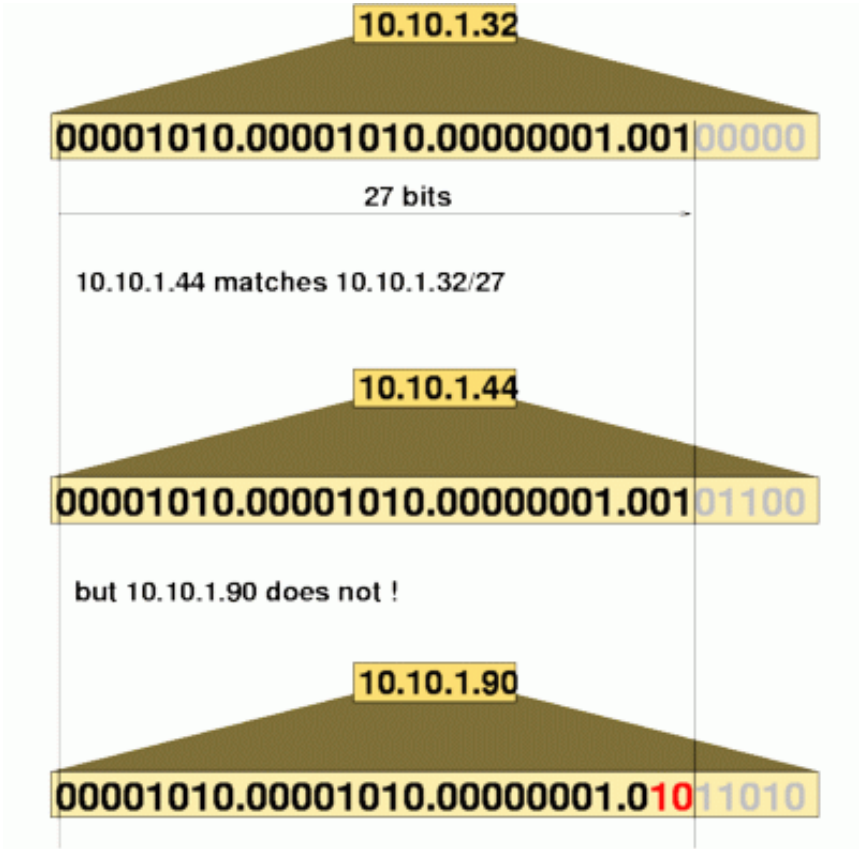
Any prefix may have a more specific prefix covering only a part of its range or it may be a more specific prefix to an even larger, less specific one.

The most specific entry in the routing table for a destination address must win.

# Network Layer - CIDR

Wikipedia CIDR graphs:



# Network Layer - Routing Table

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.

A trie (reTRIEval 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.

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.

# Network Layer - Routing Table

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 until root node of the tree is reached again.

With an entry count of 173,000 and backtracking the PATRICIA trie gets very inefficient on modern CPU's and SMP.

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.

# Network Layer - Routing Table

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.

Other more efficient algorithms exist. For example LC-trie.

# Network Layer - Routing Table

FreeBSD 7-CURRENT will implement a different but very simple, yet very efficient routing table algorithm.

It 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.

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 is cache prefetching, high memory bandwidth and fast computations.

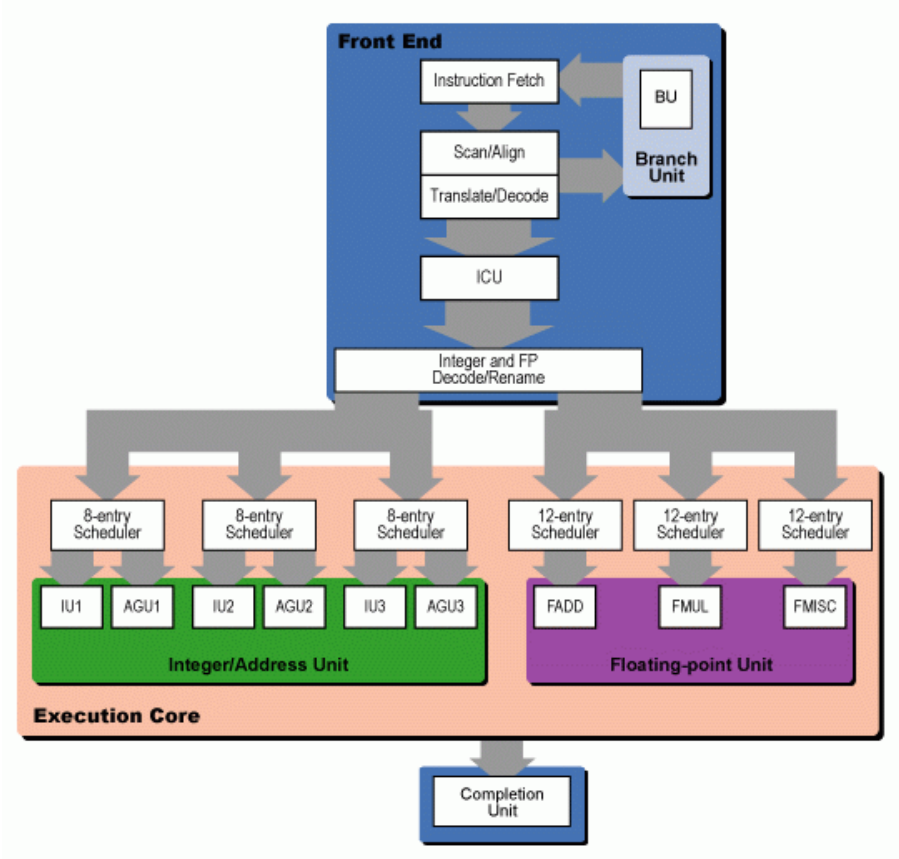
# Network Layer - Routing Table

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.

[[trie graph]]

# Network Layer - Routing Table

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.



# Network Layer - Routing Table

The footprint of each entry is very small and the entire table fits into approximately the same amount of space as the LC-trie. 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 a tables a write lock is required to serialize all changes.

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.

# Transport Layer - PCB Lookup

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 no match the packet gets dropped and an ICMP error message is sent back.

For TCP packets, now called segments, the socket lookup is complicated.

# Transport Layer - TCPCB

To make a determination where to deliver the packet a hash table is employed.

Prior to the hash table lookup can be made the entire TCP control block list 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.

This locks out any concurrent TCP segment processing on SMP as only one CPU may handle a segment at any give point in time.

# Transport Layer - TCPCB

On one hand this is bad because it limits parallelism.

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.

# Transport Layer - TCPCB

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 to 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.

# Transport Layer - TCP Reassembly

TCP guarantees 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.

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.

# Transport Layer - TCP Reassembly

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).

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 received later segments.

# Transport Layer - TCP Reassembly

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.

| 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 |

# Transport Layer - TCP Reassembly

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.

# Transport Layer - TCP Reassembly

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.

Tis 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.

Replicate this for a couple of connections and the entire server runs out of available memory.

We need something better!

# Transport Layer - TCP Reassembly

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.

This covers a large part of the malicious attack scenarios.

To thwart all other attacks described in research papers only the number of missing segments (holes) has to be limited.

# Transport Layer - TOE

TCP Offload Engines (TOE)

With TOE the clear disadvantage 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.

Additionally 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.

# Transport Layer - TSO

TCP segmentation offloading (TSO)

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.

# Transport Layer - TSO

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.

An implementation of TSO for FreeBSD's network stack is a non-trivial endeavor.

# Session Layer - T/TCPv2

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.

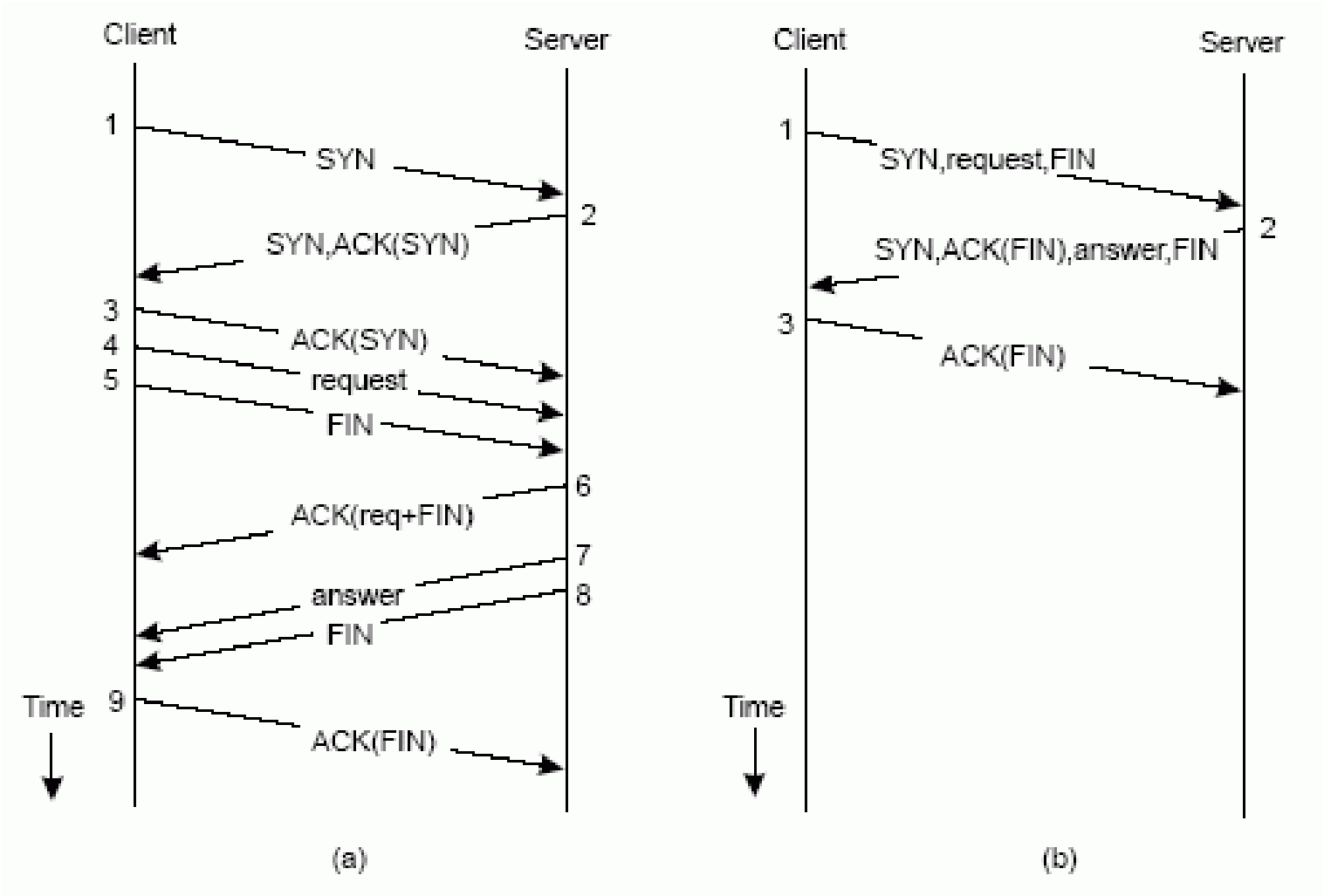
It was observed early on that the single largest latency block in short TCP connections comes from the three way handshake.

# Session Layer - T/TCPv2

T/TCP optimizes this by doing a three way handshake only the first time any two hosts communicate with each other.

Allfollowing connections send their data/request segment directly with the first SYN packet.

# Session Layer - T/TCPv2



# Session Layer - T/TCPv2

Old T/TCP implemented to RFC1644 is very weak on security and the TCP part of the implementation is complicated.

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.

# Session Layer - T/TCPv2

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.

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.

# Session Layer - T/TCPv2

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.

# Session Layer - T/TCPv2

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.

# Session Layer - T/TCPv2

The 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 - HTTP Server

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.

Along the HTTP request handling 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.

# Application Layer - Accept Filter

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.

# Application Layer - select/poll

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.

# Application Layer - kqueue

FreeBSD has introduced an event driven mechanism called kqueue(2) to overcome this limitation.

With kqueue the application registers a kernel event on the socket file descriptor and specifies which events it is interested in.

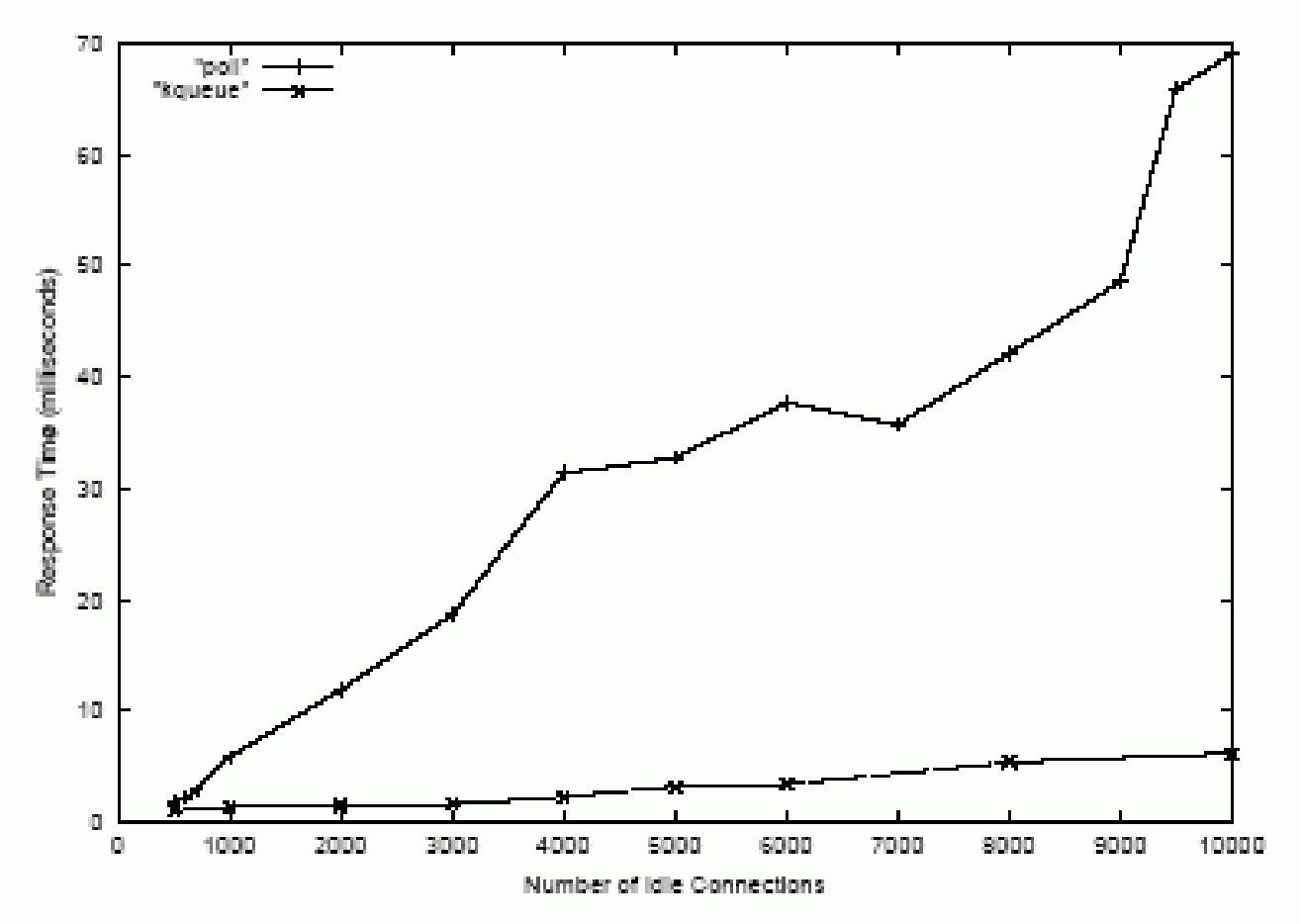
The registered event is active until it is cancelled (or the socket goes away).

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.

# Application Layer - kqueue

Poll vs. kqueue:



# Application Layer - stat

Once the request has been 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.

# Application Layer - sendfile

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.

Here again the sendfile(2) system call may block if not all file content is in the file system or buffer cache causing.

# Application Layer - sendfile

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 the references in my paper.

# 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

http://people.freebsd.org/~andre/tcpoptimization.html

That’s it. Any questions?

Download this paper at
http://people.freebsd.org/~andre/