Scalable Kernel TCP Design and Implementation for Short-Lived Connections

Xiaofeng Lin
Sina Corporation, ZHIHU Corporation
jerrylin.lxf@gmail.com

Yu Chen
Department of Computer Science and Technology, Tsinghua University
yuchen@mail.tsinghua.edu.cn

Xiaodong Li
Sina Corporation
xiaodong2@staff.sina.com.cn

Junjie Mao Jiaquan He Wei Xu Yuanchun Shi
Department of Computer Science and Technology, Tsinghua University
maojj12@mails.tsinghua.edu.cn, objectkuan@gmail.com, weixu@tsinghua.edu.cn, shiyc@tsinghua.edu.cn

Abstract

With the rapid growth of network bandwidth, increases in CPU cores on a single machine, and application API models demanding more short-lived connections, a scalable TCP stack is performance-critical.

Although many clean-state designs have been proposed, production environments still call for a bottom-up parallel TCP stack design that is backward-compatible with existing applications.

We present Fastsocket, a BSD Socket-compatible and scalable kernel socket design, which achieves table-level connection partition in TCP stack and guarantees connection locality for both passive and active connections. Fastsocket architecture is a ground up partition design, from NIC interrupts all the way up to applications, which naturally eliminates various lock contentions in the entire stack. Moreover, Fastsocket maintains the full functionality of the kernel TCP stack and BSD-socket-compatible API, and thus applications need no modifications.

Our evaluations show that Fastsocket achieves a speedup of 20.4x on a 24-core machine under a workload of short-lived connections, outperforming the state-of-the-art Linux kernel TCP implementations. When scaling up to 24 CPU cores, Fastsocket increases the throughput of Nginx and HAProxy by 267% and 621% respectively compared with the base Linux kernel. We also demonstrate that Fastsocket can achieve scalability and preserve BSD socket API at the same time. Fastsocket is already deployed in the production environment of Sina WeiBo, serving 50 million daily active users and billions of requests per day.

Categories and Subject Descriptors D.4.4 [*Operating Systems*]: Communications Management—Network communication

Keywords TCP/IP; multicore system; operating system

ASPLOS '16 April 2–6, 2016, Atlanta, Georgia, USA.

DOI: http://dx.doi.org/10.1145/2872362.2872391

1. Introduction

Recent years have witnessed a rapid growth of mobile devices and apps. These new mobile apps generate a large amount of the traffic consisting of short-lived TCP connections [39]. HTTP is a typical source of short-lived TCP connections. For example, in Sina WeiBo, the typical request length of one heavily invoked HTTP interface is around 600 bytes and the corresponding response length is typically about 1200 bytes. Both the request and the response only consume one single IP packet. After the two-packet data transaction is done, the connection is closed, which closely matches the characteristics of short-lived connection. In this situation, the speed of establishment and termination of TCP connections becomes crucial for server performance.

As the number of CPU cores in one machine increases, the scalability of the network stack plays a key role in the performance of the network applications on multi-core platforms. For long-lived connections, the metadata management for new connections is not frequent enough to cause significant contentions. Thus we do not observe scalability issues of the TCP stack in these cases. However, it is challenging to reach the same level of scalability for short-lived connections as that of long-lived connections, since it involves frequent and expensive TCP connection establishment and termination, which result in severe shared resource

contentions in the TCP Control Block (TCB) and Virtual File System (VFS).

TCB Management The TCP connection establishment and termination operations involve managing the global TCBs which are represented as TCP sockets in the Linux kernel. All these sockets are managed in two global hash tables, known as the *listen table* and the *established table*. Since these two tables are shared system-wide, synchronization are inevitable among the CPU cores.

In addition, the TCB of a connection is accessed and updated in two phases:

1. Incoming packets of the connection are received in the interrupt context through network stack on the CPU core that receives the NIC (Network Interface Controller) interrupt.
2. Further processing of incoming packets payload in user-level application and transmission of the outgoing packets are accomplished on the CPU core running the corresponding application.

It is ideal that for a given connection, both phases are handled entirely on one CPU core, which maximizes CPU cache efficiency. We refer to this property as *connection locality*. Unfortunately, in the current Linux TCP stack, these two phases often run on different CPU cores, which seriously affects scalability.

VFS Abstraction A socket is abstracted by VFS and exposed to user-level applications as a socket *File Descriptor* (FD). As a result, the performance of opening and closing a socket FD in VFS has a direct impact on the efficiency of the TCP connection establishment and termination. However, there are severe synchronization overheads in processing VFS shared states, e.g., `inode` and `dentry`, which results in scalability bottlenecks.

Serious lock contention are observed on an 8-core production server running HAProxy as an HTTP load balancer for WeiBo to distribute client traffic to multiple servers. From profiling data collected by *perf*, we observed that `spin_lock` consumes 9% and 11% of total CPU cycles in TCB management and VFS respectively. Even worse, due to contention inside the network stack processing, there is a clear load imbalance across all CPU cores, though packets are evenly distributed to these CPU cores by NIC.

Production Environment Requirements In production environment, such as data center, the network server/proxy applications have three fundamental requirements:

1. Compatibility of TCP/IP related RFC: Different applications follow different TCP/IP related RFCs. As a result, the TCP/IP implementation in production environment also need to support commonly accepted RFCs as far as possible.
2. Security: TCP may be attacked in a variety of ways(DDoS, Connection hijacking, etc.). These attacks result of a thorough security protection of TCP [13] in OS kernel.
3. Resource Isolation and Sharing: TCP/IP stack needs to support diverse existed NIC hardware resources, and should isolated or share data with other name space or sub-systems (disks, file systems, etc.)

Extensive researches [12, 21, 22, 27, 30, 38] are focused on system support for enabling network-intensive applications to achieve performance close to the limits imposed by the hardware. Some work proposes a brand new I/O abstraction to make the TCP stack scalable [21]. However, applications have to be rewritten with the new API. Recent research projects [12, 22, 30] attempt to solve the TCP stack scalability problem using modified embedded TCP stack lwIP [26] or custom-built user-level TCP stack from the ground up. Though these design can improve the performance of network stack, it does not fully replicate all kernel's advanced networking features (Firewall/Netfilter, TCP veto, IPsec NAT and other vulnerabilities defenses for security, TCP optimizations for wireless net, cgroup/sendfile/splice mechanism for resource isolating or sharing, etc.) which are heavily utilized by our performance-critical applications.

Therefore, currently it remains an open challenge to completely solve the TCP stack scalability problem while keeping backward compatibility and full features to benefit existing network applications in the production environment.

In this paper, we present Fastsocket to address the aforementioned scalability and compatibility problems in a backward compatible, incrementally deployable and maintainable way. This is achieved with three major changes: 1) partition the globally shared data structures, the listen table and established table; 2) correctly steer incoming packets to achieve connection locality for any connection; and 3) provide a fast path for socket in VFS to solve scalability problems in VFS and wrap the aforementioned TCP designs to fully preserve BSD socket API.

In the production environment, with Fastsocket, the contention in TCB management and VFS is eliminated and the effective capacity of HAProxy server is increased by 53.5%. Our experimental evaluations show that in short-lived connection benchmark of web servers and proxy servers handling short-lived connections, Fastsocket outperforms base Linux kernel by 267% and 621% respectively.

This paper makes three key contributions:

1. We introduce a kernel network stack design which achieves table-level partition for TCB management and realizes connection locality for arbitrary connection types. Our partition scheme is complete and naturally solves all potential contentions along the network processing path, including the contentions we do not directly address such as timer lock contention.
2. We demonstrate that BSD socket API does not have to be an obstacle in network stack scalability for commodity hardware configuration. The evaluation using real-world applications and benchmarks shows throughput

with Fastsocket can scale up to 20.4x on a 24-core machine with 10G NIC, using the BSD socket API. The results suggest that there is no scalability reasons to give up the BSD socket API. Moreover, this compatibility ensures that existing network applications do not need modifications to use Fastsocket.

3. We demonstrate that with moderate modifications in kernel, we can build a highly scalable kernel network stack. By integrating into the kernel framework, our design can preserve robustness and all features of kernel network stack, which makes it a practical scheme for production environment deployment. Fastsocket follows this idea and has been massively deployed in the Sina WeiBo production environment.

The rest of this paper is organized as follows. In Section 2, we discuss the challenges in scaling TCP stack while being backward compatible in more detail. We describe the Fastsocket design in Section 3. Section 4 evaluates Fastsocket in the production environment and with several benchmarks as well. Section 5 presents related work and we conclude in Section 6.

2. Challenges

In this section, we focus on the scalability bottlenecks: TCB management and VFS abstraction. We analyze the challenge to completely solve these bottlenecks and parallelize TCP stack while keeping backward compatibility.

2.1 Bottleneck in TCB Management

Management of global TCBs is a major scalability bottleneck for the TCP stack on multi-core platforms. Due to mutual exclusion requirement when operating these global TCBs, locks are widely used, leading to potential lock contentions when multiple CPU cores need to update the TCBs (represented as sockets in Linux) concurrently.

Global Listen Table

While establishing passive connections with concurrent clients requesting the same service port, a global listen socket is used to set up connections on all CPU cores, which is a major scalability bottleneck.

Both Linux `SO_REUSEPORT` [23] and Megapipe [21] address this scalability problem using socket-level partition. For example, in `SO_REUSEPORT`, each application process creates a private copy of the original listen socket and these copies are linked into a single bucket list in the listen table. When handling new connections, kernel traverses the list and randomly selects a listen socket copy from all others in `NET_RX` SoftIRQ . Application processes accept new connections from its own listen socket copy only. As a result, they can remove the global listen socket bottleneck based on this socket-level partition. However, this solution hits another scalability problem. Though in `NET_RX` SoftIRQ kernel can use multiple listen sockets to avoid intensive contention, it has to *traverse* the bucket list to choose a listen socket for any connection request. In this case, the complexity of finding a listen socket is *O(n)*, where *n* is number of CPU core in the server. Therefore, when the number of CPU cores increases, the cost of matching a listen socket becomes serious enough to be a bottleneck. Our benchmarks with `SO_REUSEPORT` show that the function that matches a listen socket, `__inet_lookup_listener()`, consumes only 0.26% CPU cycles on a single core machine, but the overhead soars to an average of 24.2% per-core with 24 CPU cores.

Although Megapipe has a different implementation, it uses the same socket-level partitioning design as `SO_REUSEPORT` and suffers from the same problem.

It may be worth attempting to use a naive table-level partition scheme, in which each CPU core has its own local listen table containing the copied listen socket to avoid searching the bucket list with a *O(n)* complexity. However, this partition breaks the TCP stack. For example, if one process in the application crashes, the operating system will destroy the copied listen socket created by the process. In this case, there is no copied listen socket in the local listen table of that certain CPU core. When a `SYN` packet is delivered to the CPU core, it cannot match any listen socket in the local socket table and the connection request from client would be rejected with a `RST` packet, even if other application processes (on other CPU cores) are available to handle the client connection, which breaks the TCP robustness.

Global Established Table

A new socket is created and added into the established table to represent an established connection for both passive and active connections. Linux currently adopts per-bucket locks to synchronize concurrent modifications to the established table shared by multiple CPU cores. This locking granularity works fine on a 8-core machine serving hundreds of concurrent connections. However, the global design of established table will inevitably lead to considerable lock contentions when both the CPU cores and concurrent connections in a single server grow rapidly. Therefore, lock granularity refinement is just an optimization but not a thorough solution to the scalability problem.

Naive partition of the global established table into per-core local tables does not work, because there is no guarantee that in `NET_RX` SoftIRQ any incoming packet is always processed on the same CPU core that has the corresponding socket in the local established table. As a result, it is possible that the socket is created by one CPU core and inserted into the local table of the CPU core while a subsequent packets of the connection arrives on the other CPU core. In this condition, the packet would be rejected and the TCP connection breaks with this naive partition design.

Through the above analysis, it is still challenging to realize a *full partition of TCB management*, that is when man-

aging any socket, no lock is ever contended and the cost is constant regardless of number of CPU cores.

2.2 Lack of Connection Locality

Incoming and outgoing packets in a single connection can be handled by two different CPU cores, which causes CPU cache bouncing and performance degradation. Therefore, for maximum scalability, it is desirable to achieve *complete connection locality*, i.e., all activities for a given connection are always handled on the same CPU core, including both passive and active connections.

It is worth noting that complete connection locality is not only a performance optimization, but also the key prerequisite to partition the established table, as we discussed in the previous section. If we can ensure a connection is always handled on the same CPU core, we naturally solve the established table partition problem.

The *completeness* is the real challenge, which has not been solved by various optimizations to *improve* connection locality in a best-effort way that only provide a probabilistic guarantee.

For passive connections, Affinity-Accept [29] and Megapipe [21] achieves connection locality by making the application accept new connections on the CPU core where the application is running. Unfortunately, this design does not apply to active connections that are heavily used in proxy and RPC (Remote Procedure Call) applications. This is because active connections are initiated in the applications which has no control over selection of the CPU core the connection will be processed later in `NET_RX` SoftIRQ .

To improve connection locality, RFS (Receive Flow Deliver) was introduced into Linux [6, 7]. RFS introduces a table to track connections in the system. In the table, RFS records on which CPU core each connection is processed in application and delivers incoming packets according to the table. However, managing the table introduces considerable overhead and complete connection locality cannot be guaranteed due to the table size limit.

Modern NIC provides advanced features such as FDir (Flow Director) [9] to deliver incoming packets intelligently. FDir works in two modes: ATR (Application Target Routing) or Perfect-Filtering. ATR works similarly to RFS but differs in that ATR samples outgoing packet and maintains the connection-CPU-mapping table in the NIC. Since the mapping is based on sampling and the size of hardware table is limited, it is actually a best-effort solution instead of a complete solution. With Perfect-Filtering, users can program NIC to a limited extent to decide which CPU core to deliver an incoming packet to, based on the IP address and TCP port of the packet. However, it is impractical to use Perfect-Filtering to realize active connection locality, as there lacks a general pattern of the IP address and TCP port we can leverage to program the delivery policy to the NIC.

Even if there is a design realizing active connection locality, in order to achieve complete connection locality, the design cannot break the passive connection locality, which makes it even more challenging to implement complete connection locality.

2.3 Additional Overhead for Compatibility

Socket has an abstraction in VFS as a special file type and VFS associates each socket with corresponding `inode` and `dentry` data structures in the same way as other file types. As discussed in [14], when sockets are frequently allocated, VFS spends much effort to manage the globally visible `inodes` and `dentries`, which involves expensive synchronization.

Mainline Linux kernel and research[14] use fine-grained locking mechanism,sloppy counter,lock-free protocol, etc. to improve the efficiency of VFS synchronization. However, we believe it is unnecessary for socket to inherit such overheads, even if they are becoming smaller, as `inode` and `dentry` are actually not being used by socket at all, because of the fundamental differences between networking and a disk file.

Recent researches have proposed two kinds of new designs to address the scalability problem of VFS for socket and TCP stack. For example, a user-level TCP stack [22] and a new IO API design [21]. These new designs avoid synchronization at the cost of advanced TCP stack features and backward compatibility.

The user-level TCP stack, mTCP [22], or modified embedded TCP stack, lwIP [12, 30], used by rescent research works, eliminates VFS overhead and bypasses all the complexity of kernel to achieve high performance. However, the performance gain is not free. First, these TCP stacks implement a light version of TCP stack. Many advanced features such as namespace and firewall, which are very useful in practice, are not implemented. Overtime, these stacks could replicate the kernel functions, but it will be a very complex and expensive development process. Second, the APIs provided by these network stacks are not fully compatible with BSD socket, VFS and RFCs, leading to expensive code changes in applications. Finally, to enable a NIC to work with these TCP stacks, we can no longer share the NIC with the kernel, preventing other "normal" applications from running on the same NIC port. All these limitations make it impractical for us to deploy in production environment.

Megapipe [21], on the other hand, introduces a new I/O API which completely bypasses VFS and maintains the in-kernel advantages that user-level TCP stack does not provide. However, Megapipe is no longer compatible with socket API. To deploy Megapipe, we need to modify all existing applications and tools. Some of the modifications are nontrivial as Megapipe presents a different network programming model. The incompatibility severely limits the real-world applicability of Megapipe.

Our goal is to redesign a kernel-level TCP stack that keeps compatibility to the BSD socket API while achieving

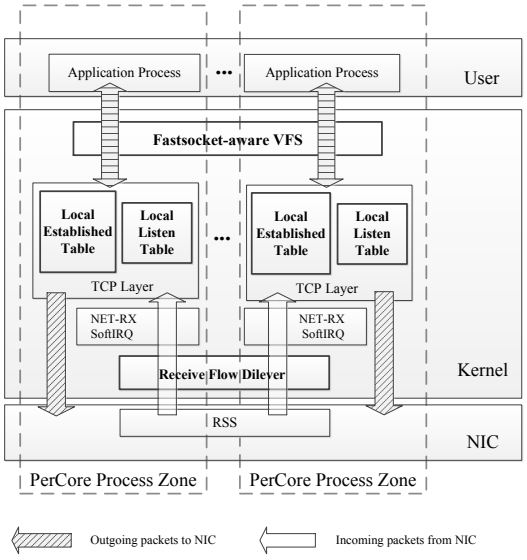

Figure 1: Fastsocket Architecture

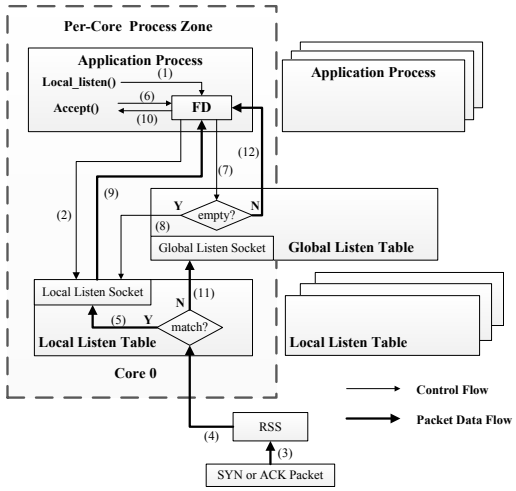

Figure 2: Local Listen Table

high scalability and low overhead, which is the genuine demand for large-scale production environment.

3. Fastsocket Design

3.1 Architecture Overview

Figure 1 presents an architecture overview of Fastsocket. Fastsocket provides three components, the partitioned TCB data structures (the Local Listen Table and the Local Established Table), Receive Flow Deliver (RFD), and Fastsocket-aware VFS, which are highlighted with bold box in Figure 1. The three modules cooperate to provide Per-Core Process Zones in which all activities for a given connection, from NIC interrupts up to application accesses, are executed on a single CPU core with minimum cross-core synchronization.

To build the Per-Core Process Zone, we have made improvements to the shared data structure, the connection-core binding and the VFS abstraction implementation:

1. Fastsocket fully partitions the data structure and management of both global TCB tables. Thus, when `NET_RX` SoftIRQ reaches the TCP layer, Fastsocket uses per-core Local Listen Table and per-core Local Established Table for a full partitioning of TCB management.
2. Before an incoming packet enters the network stack, Fastsocket uses Receive Flow Deliver to steer the packet to a proper CPU core where the corresponding local table is managed and its relating application process is running. This way, Fastsocket achieves maximum connection locality and minimize CPU cache bouncing.
3. The kernel VFS layer is the key to the socket API abstraction and compatibility. Fastsocket-aware VFS bypasses unnecessary and lock-intensive VFS routines and uses socket-special fast path instead to eliminate scalability bottlenecks, while retaining enough states to provide all socket API functionalities.

The remaining of this section describes each of the three Fastsocket component designs in detail.

3.2 Full Partition of TCB Management

In this section, we describe ways to use Local Listen Table and Local Established Table to implement table-level partitions of the listen sockets and established sockets management.

3.2.1 Local Listen Table

We propose Local Listen Table to realize table-level partition for listen sockets while guaranteeing the robustness of the TCP stack. In addition, passive connection locality is also achieved with Local Listen Table.

In the kernel TCP setup stage, Fastsocket allocates a *local listen table* for each CPU core and maintains the original *global listen table* for robustness.

On startup, the first process of a server application typically `listen()`s on a certain TCP port to wait for client requests. The kernel correspondingly creates a listen socket in the global listen table. Then the server forks multiple processes and binds them with different CPU cores. These child processes inherit the listen socket and are ready to accept new connections and process accepted connections in parallel.

With Fastsocket, each process invokes `local_listen()` to inform the kernel that the process wants to handle incoming connections from the CPU core to which it has been bound (1). As Figure 2 shows, for the process bound on CPU core 0, a new listen socket is copied from the original listen socket and inserted into the local listen table of CPU core 0 (2). We refer to the copied listen socket as the *local lis-*

ten socket and the original listen socket as the *global listen socket*.

Steps (3) to (6) and (8) to (10) in Figure 2 illustrates the fast (normal) path of setting up a passive connection using local listen table, and step (7), (11) and (12) illustrate slow (abnormal) path to handle the fault condition.

Fast Path

When a `SYN` packet comes in (3) and it is delivered to CPU core 0 by RSS, the kernel searches the local listen table of CPU core 0 to match a local listen socket (4). Without any failures, the local listen socket which is previously inserted by `local_listen()` will be matched (5). After the three-way handshake is done, a ready connection is placed in the accept queue of the local listen socket. When the process bound on core 0 issues `accept()` for new connections (6), the kernel first checks the accept queue of the global listen socket (7). Under normal operation, the accept queue is always empty. This check is done without locking, as we will explain later in this section. Then the kernel will check the local listen table of core 0 for any ready connection (8). Naturally, the previously ready connection can be found (9) and returned to application process (10).

Therefore, with a table-level partition of Local Listen Table, every operation (except for the lock-free check) is on the local table, which eliminates the synchronization bottleneck during the establishment of a passive connection.

As a by-product, we also achieve passive connection locality as all processing (in application process and the kernel interrupt context) of any passive connection runs on the same CPU core.

Slow Path

Application failures complicate things a bit, as we have described in Section 2.1. Under certain unusual circumstance, it is possible that the copied listen socket in the local listen table of certain CPU core is missing.

In Fastsocket, Figure 2 shows, when a `SYN` packet cannot match a local listen socket in the local listen table, the kernel will set up a new connection with the global listen socket in the global listen table (11). When any application process calls `accept()` (6), the ready connection in the global listen socket will be found since the accept queue of the global listen socket is checked first (7). The kernel can then `accept()` the connection from the global listen socket to the application process, just what the legacy TCP stack does (12).

As we have mentioned before, we need to check the accept queue of the global listen socket first, prior to accepting a new connection from the local listen socket. This is because on a busy server, there are always new connections in the local listen socket. If we check the local listen socket first, we will keeping processing connections in the local listen socket and starve the connection in the global listen socket.

In addition, no locking is needed when checking the global listen socket, since it is done by checking whether the accept queue pointer is `NULL` using a single atomic read operation. If there is a new connection in the accept queue (very rare), we have to lock the global listen socket. When the socket is locked, the accept queue is checked again and we accept the new connection from the global listen socket. Though locking is involved in the slow path, this circumstance rarely happens in practice, and thus does not cause lock contentions.

In summary, Fastsocket solves any scalability bottlenecks in passive connection establishment with Local Listen Table, and achieves passive connection locality while keeping robustness of TCP under application failures.

3.2.2 Local Established Table

We propose Local Established Table to partition the established table for locally accessing established sockets.

Operations of Local Established Table are designed as follows:

- Fastsocket allocates a *local established table* for each CPU core when kernel initializes network stack.
- New established sockets are inserted into the local established tables.
- In `NET_RX` SoftIRQ , the kernel checks the local established table to match an established socket for any incoming packet.

As described in Section 2.1, we need to make sure that for any connection, inserting and matching the established socket are carried out on the same CPU core to make this design work. Fastsocket provides the guarantee we will describe in Section 3.3.

With Local Established Table, established sockets management is fully partitioned across the CPU cores and this design eliminates scalability problems in the established table regardless of the number of CPU cores in a single machine.

At this point, Fastsocket has achieved table-level partition for both listen table and established table and a full partition of TCB management is achieved.

3.3 Complete Connection Locality

As shown in previous section, Local Listen Table design realizes the passive connection locality. In this section, we propose Receive Flow Deliver (RFD) to solve the active connection locality problem described in Setion 2.2 to achieve the complete connection locality.

Build Active Connection Locality

The key insight is that the kernel can encode the current CPU core id into the source port when making an active connection request. To achieve this, we use a hash function $hash(p)$ that maps ports to CPU cores. When the application running

on CPU core c attempts to establish an active connection, RFD chooses a port p_{src} so that $c = hash(p_{src})$. Upon receiving a response packet, RFD picks the destination port of the received packet p_{dst} which is the port RFD previously chosen, determines which CPU core should handle the packet by $hash(p_{dst})$, and steers the packet to the selected CPU core, if this is not the CPU core currently handling the packet. In this way, RFD guarantees that each active connection is always processed by the same CPU core, which eliminates the non-local connection problem.

Retain Passive Connection Locality

Passive connections and active connections can exist in one machine simultaneously, and incoming traffic consists of both passive incoming packets and active incoming packets. Before decoding the CPU core id from the destination port of incoming packets, RFD has to distinguish the two incoming packet categories since the hash function is designed for active connections and should only be applied to active incoming packets. Otherwise, RFD would break the passive connection locality from Local Listen Table.

To determine whether an incoming packet is a passive incoming packet or an active incoming packet, RFD applies the following rules in order:

1. If the source port of an incoming packet is in the range of the well-known (less than 1024) ports, it should be an active incoming packet. We assume the kernel never picks a port within the well-known ports as the source port under normal conditions.
2. If the destination port is a well-known port, it is a passive incoming packet for the same reason.
3. (optional) If neither of the previous condition meets, we see whether the packet can match a listen socket. If so, it indicates that it is a passive incoming packet because it is not allowed to start an active connection with a source port that was previously listened. Otherwise, the packet is an active incoming packet.

The classification work could be done within the first two quick checks when the application is serving on well-known ports, which is usually the case. A precise classification, if required, can be guaranteed by applying the third rule directly(the first two rules should be skipped in this case).

Leverage Hardware Features

RFD can work correctly without any hardware NIC features. However, as synchronization is required when RFD needs to steer a packet to a different CPU core, it is beneficial for RFD to take advantage of advanced NIC features such as FDir and offload the steering work to the NIC.

As Section 2.2 describes, there are two modes in FDir: Perfect-Filtering and ATR. When using Perfect-Filtering, RFD may program the hash function into NIC in order to offload the steering work completely to hardware. When using ATR mode, the majority of incoming packets are delivered to the right CPU core thanks to the sampling of NICs and the rest of them are left to RFD to handle. Thus, RFD can be used either with Perfect-Filtering for maximum performance benefits at a cost of extra NIC configuration, or with ATR to enhance connection locality of the system with minimal maintenance efforts, and coexists with other applications requiring the ATR mode.

Our RFD Implementation

To choose a proper hash function, we need to consider the capability of the underlying NIC hardware and the software overhead for extra packet steering work in RFD.

In our implementation of Fastsocket, we use the following hash function:

$hash(p)$ = `p & (ROUND_UP_POWER_OF_2(n) - 1)`

where `ROUND_UP_POWER_OF_2(x)` returns the next power of two of *x* and *n* is the number of CPU cores we have.

This function can be easily programmed into the FDir in Perfect-Filtering mode which supports bit-wise operations only.

We can introduce some randomness to improve security by randomly selecting the bits used in the operation. It helps prevent malicious attacks that try to make the server process all connections on the same CPU core. We can also leverage existing efforts on defending such attacks [5, 25] on top of Fastsocket. The defensing work could be left to the dedicated network security devices as well. The security solutions are beyond the scope of this paper.

With Local Listen Table and Receive Flow Deliver, Fastsocket has achieved complete connection locality.

3.4 Fastsocket-aware VFS

In some operating systems, such as Linux, socket is managed by VFS abstraction among other file types. However, sockets are fundamentally different from files on disk [21].

1. **Sockets are not on disk.** Due to the slow speed of disks, `dentry` and `inode` are used as memory caches for the metadata of files on the slow disk. However, sockets exist in memory for their entire lifecycle and cannot benefit from the caching mechanism for which `dentry` and `inode` are designed.
2. **Directory path is never used to identify socket.** Usually, directory path is used to identify the disk files, and internally the kernel maps the directory path to a `dentry` and its corresponding `inode` to find the file. However, sockets are accessed directly through the `private_data` field of file structures. As a result, `dentry` and `inode` are not being used to identify sockets.

Therefore, we can see that `dentry` and `inode` are functionally useless for sockets, but still associated with sockets to be compatible with VFS abstraction. As discussed in 2.3,

the useless `dentry` and `inode` introduce tremendous synchronization overhead when sockets are allocated frequently.

We propose Fastsocket-aware VFS to tackle the VFS scalability bottleneck for sockets while keeping the Fastsocket compatible with legacy VFS.

Avoid Unnecessary Overhead Fastsocket-aware VFS provides a fast path in the VFS processing for sockets to avoid the unnecessary overhead. Specifically, Fastsocket-aware VFS skips most initialization/destruction work of `dentry` and `inode` when creating/destroying a socket, which involves acquiring several locks and managing the `dentry` and `inode` in various tables and links.

Keep Compatibility We cannot remove `dentry` and `inode` from socket completely, as the remove will break the compatibility with legacy sockets. For example, widely used system tools such as *lsof* and *netstat* will fail, since they access the corresponding socket status through */proc* file system, which relies on `dentry` and `inode` (but does not require them to be completely initialized). Fastsocket-aware VFS only keeps the necessary states and functions of `dentry` and `inode` to support */proc* file system, which is a small price to pay to achieve compatibility.

In short, Fastsocket-aware VFS internally customizes VFS for sockets to improve scalability while externally keeping VFS compatibility for socket applications and system tools.

4. Evaluation

To evaluate the performance and scalability of Fastsocket, we focus on the following two questions in this section:

1. Does Fastsocket benefit real applications?
2. How does Fastsocket scale, compared to the state-of-the-art Linux TCP kernel stack implementations, especially for short-lived TCP connection handling?

To answer the first question, we evaluate Fastsocket on a 8-core server running HAProxy as HTTP load balancer in a production environment and we observe a 53.5% improvement in the effective capacity by deploying Fastsocket. In addition, we also carry out an experimental evaluation with Nginx and HAProxy, the two most important component in our infrastructure. The results show that their throughputs with Fastsocket outperform the base Linux by 267% and 621% respectively on a 24-core machine.

For the second question, The results show that the throughput with Fastsocket on a 24-core system scales to 20x while Linux 3.13 with `SO_REUSEPORT` [23] achieves only 12x. In addition, we collect statistics on locking and L3 shared cache missing rate in order to illustrate how and why each individual design benefits the overall performance.

With the excellent scalability performance of Fastsocket from the evaluation result, we demonstrate that BSD socket API is no longer the scalability obstacle for TCP stack.

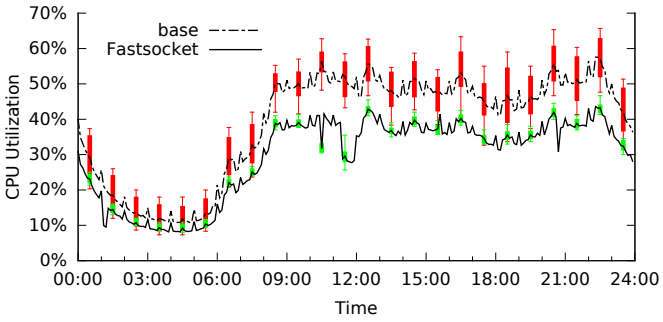

Figure 3: CPU utilization of two 8-core HAProxy servers in the Sina WeiBo production environment. One is with Fastsocket and the other is not. The upper line presents the average CPU utilization of the server without Fastsocket in one day and the lower line is with Fastsocket. We adopt a box plot [37] to visually convey the spread of 8 CPU core utilization for the two servers, which are sampled every hour.

4.1 Experiment Setup

Our testing environment consists of three machines, each of which has two 12-core Intel Xeon E5 2697 v2 processors, 64GB memory and a dual-port Intel 82599 10GE NIC. RSS is enabled on the NICs if FDir is not explicitly configured. The number of NIC queues is configured according to the number of CPU cores involved. We configure NIC interrupt affinity and XPS (Transmit Packet Steering) [8] to assign each RX and TX queue to one CPU core. As the baseline kernel has trouble in fully utilizing CPU resources with a single listen socket, all benchmarks listen on different IP addresses but on the same port 80.

All benchmarks fork multiple processes per server application and pin each process to a different CPU core. We use *connections per second* to measure throughput, as our main concern is the cost associated with connection establishment and termination. Unless otherwise stated, *http_load* [2] is used as the HTTP workload generator.

In order to saturate the benchmark servers with Fastsocket, we have to deploy Fastsocket on the clients and backend servers to increase their throughput to the same level.

4.2 Application Performance

4.2.1 Production Environment Evaluation

We have deployed Fastsocket on production servers running HAProxies in the Sina WeiBo production system. We collected the CPU utilization on two servers handling the same amount of requests (ensured by the load balancer) during the same period of time. Each server has two 4-core CPUs and 1GE NIC (without FDir support). *HTTP Keep-alive*[3] is disabled on HAProxy. One server is Fastsocket-enabled and the other is not. Figure 3 presents the results.

When serving client requests, we need to guarantee some service-level agreement (SLA), such as the 99.9th percentile response latency. In our production environment, we need to keep the CPU utilization under a certain threshold, in our case, 75%, to achieve the SLA. If any CPU core reaches the threshold, the workload handled by that CPU core will suffer a long latency, causing a latency SLA violation, even if other CPU cores are still idle. We describe this situation as the server has reached its *effective capacity*. The effective capacity is determined by the *most utilized* CPU core. Therefore, the load balance among different CPU cores and avoiding a "hot" CPU core is an important performance optimization.

As Figure 3 shows, without Fastsocket, the average CPU utilization is relatively high. Moreover, the variation of CPU utilization is very high among the 8 CPU cores (the error bar shows the max and min CPU utilization on different CPU cores). The underlying reason is using shared resources and lock contentions in the Linux kernel. For example, when multiple CPU cores try to get new connections from the shared accept queue, they need to first acquire the listen socket lock, as described in Section 2.1. The first CPU core will take the connection and there would probably be no connection left for the last CPU core when the core acquires the lock in turn. This can happen each time when kernel informs application that there are new connections ready to be accepted, which makes certain CPU cores have more connections to process and introduces the sustaining load variances among different CPU cores.

In the contrast, with Fastsocket, the average CPU utilization is considerably reduced because of the lower overhead in lock contention. Even better, the utilization of all CPU cores is perfectly balanced since Fastsocket eliminates shared resources and lock contentions in TCP stack.

Take a busy time 18:30 as an example. Without Fastsocket, average CPU utilization is 45.1%, while for the server with Fastsocket, the number is 34.3%, revealing that Fastsocket can improve CPU efficiency by 31.5% in handling the same amount of real traffic. Without Fastsocket, the CPU utilization varies from 31.7% to 57.7%, and with Fastsocket, the utilization of all CPU cores is very close to each other, ranging from 32.7% to 37.6%. We reasonably assume that the CPU consumption of a server is linear with the traffic it is supporting. Thus, the effective capacity and the utilization of the most utilized CPU core are inversely proportional. As a result, Fastsocket has improved effective capacity by

$$\frac{(37.6\%)^{-1} - (57.7\%)^{-1}}{(57.7\%)^{-1}} = 53.5\%$$

In addition, these production servers are relatively old machines that only have 8 CPU cores. The scalability problem only gets worse with the increasing number of CPU cores in the very near future. Fastsocket will become more crucial in the new multi-core machines. Nonetheless, the 53.5% performance boost by Fastsocket is still a big cost-saving in the production environment.

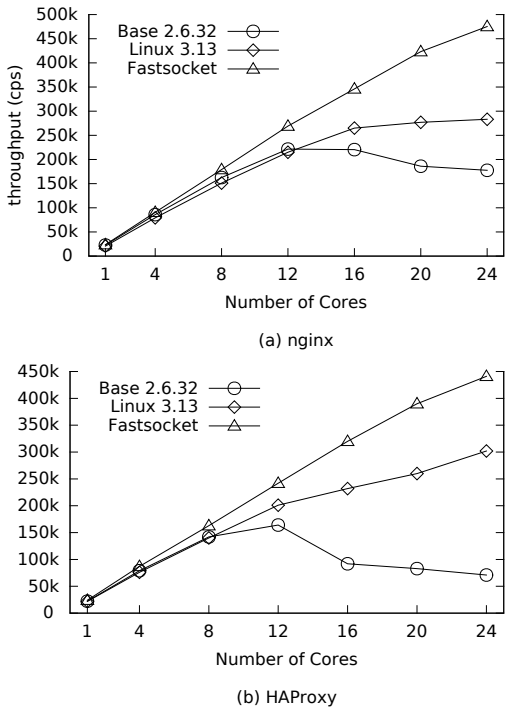

Figure 4: Connections per second throughput of the Nginx and HAProxy tests

4.2.2 Nginx

We perform Nginx benchmark to evaluate Fastsocket with the real-world web server. We disable HTTP Keep-alive on Nginx to allow the emulated clients to establish multiple short-lived connections. *Http_load* fetches a 64 bytes static file from Nginx with a concurrency of 500 multiplied by the number of CPU cores. The file is cached in memory so no disk I/O is involved. As the listen socket contention is eliminated by Fastsocket, we disabled `accept_mutex` in the Fastsocket test.

The throughput of Nginx on 1 to 24 CPU cores are shown in Figure 4(a). Fastsocket with Linux 2.6.32 on 24 CPU cores achieves 475K connections per second, with a speed up of 20.0x. The throughput of Nginx with base 2.6.32 kernel increases non-linearly up to 12 CPU cores and drops dramatically to 178K with 24 CPU cores. The latest 3.13 kernel only achieves the throughput to 283K when using 24 CPU cores. Here we observe that the latest kernel does not completely solve the scalability bottlenecks, preventing it from scaling beyond 12 CPU cores. We will show the reason later in Section 4.2.4.

4.2.3 HAProxy

In addition to Nginx, we present our HAProxy benchmark, to demonstrate the active connection (to the backend servers)

performance. We setup a client running *http_load* with concurrency of 500 multiplied by number of CPU cores and a backend server sending a constant 64-byte page.

Figure 4(b) presents the throughput of HAProxy. Fastsocket outperforms Linux 3.13 by 139K connections per second and base 2.6.32 by 370K when using 24 CPU cores, though the single CPU core throughputs are very close among all the three kernels. HAProxy is a proxy application and different with Nginx in that it makes frequently active connections to backend servers. With Receive Flow Deliver, Fastsocket introduces connection locality for active connections and greatly reduces CPU cache bouncing. This local processing of connections is important since Receive Flow Deliver increases throughput by 15%, as shown in Section 4.2.4.

4.2.4 Analysis on Fastsocket Performance Improvements

Lock Contention

To understand the effect of Fastsocket on reducing lock contentions, we have run the HAProxy benchmark with 24 CPU cores for 60 seconds and used *lockstat* [4] statistics to measure lock contentions. Table 1 presents the hottest locks, their contention counts in the baseline Linux and their contention count changes after enabling Fastsocket.

For the baseline kernel, `dcache_lock` and `inode_lock` in VFS suffer from the most severe contention and Fastsocket-aware VFS eliminates these unnecessary contentions as discussed in Section 3.4.

Socket is shared between interrupt context and process context in the TCP stacks by multiple CPU cores, therefore, a `slock` is used to protect the shared state of socket. Using Local Listen Table and Receive Flow Deliver, we achieve complete connection locality, therefore, each socket is processed only on a single CPU core, which eliminates any socket sharing across CPU cores. Thus we reduce the contentions on `slock` to 0 when enabling Local Listen Table and Receive Flow Deliver.

TCP processing involves other kernel systems, e.g., Epoll and Timer. `Ep.lock` avoids parallel socket status updating on the event ready list of epoll instance. `Base.lock` protects modifications to per-socket timers managing TCP timeouts. They suffer the same `slock` contention problem, since they are in the critical path of TCB processing. When Fastsocket has achieved complete connection locality, these lock contentions are naturally eliminated as Table 1 shows.

`Ehash.lock` is the per-bucket lock protecting the global established table. Contentions on this lock have been completely eliminated by adopting the Local Established Table.

After Fastsocket is deployed, locks (esp. *spinlocks*) consumes no more than 6% CPU cycles and most of the locks can be acquired without contention due to the partitioned data structure in Fastsocket. As a result, lock contention is no longer a performance bottleneck in our TCP stack.

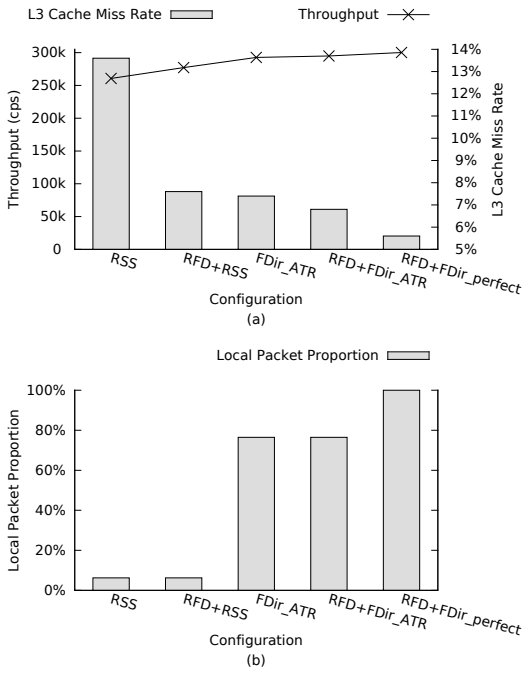

Figure 5: Throughput, L3 cache miss rate and local packet proportion with different NIC packets delivering features enabled

Connection Locality

To evaluate Receive Flow Deliver in more detail, we conduct experiments with RFD enabled or disabled, combined with different NIC packet delivering features including RSS, FDir in ATR mode (FDir_ATR) and FDir in Perfect-Filtering mode (FDir_Perfect). The experiment is carried out on a SandyBridge server with 16 CPU cores as the latest CentOS 6 does not support `ioatdma` [10] on IvyBridge yet, which would affect CPU cache usage patterns in the test. Fastsocket-aware VFS and Local Listen Table are always enabled in these experiments.

We did not perform the experiment of using FDir_Perfect without Receive Flow Deliver, because as discussed in Section 2.2, it will cause correctness problems.

We use throughput and L3 cache miss rate as metrics to illustrate the connection locality effect of RFD combined with different NIC packets delivering features. In addition, for active connections, we count the number of *local packets*, i.e. packets received from NIC and later processed in application on the same CPU core, as well as the total number of packets received. We present the percentage of local packets among all packets received to illustrate the effectiveness in preserving connection locality.

The throughput, L3 cache miss rates are presented in Figure 5(a) and local packet proportion data in Figure 5(b).

When RSS is enabled, the NIC evenly distributes incoming packets to different CPU cores, regardless of where the packets would be processed by applications. As a result, the local packet proportion is 6.2%, showing that the majority

Table 1: Lock Contention Counts (HAProxy benchmark with 24 CPU cores for 60 Seconds)

V = Fastsocket-Aware VFS, L = Local Listen Table, R = Receive Flow Deliver, E = Local Establish Table

lock	Baseline	Δ				Fastsocket
		+V	V +L	VL +R	VLR +E	
dcache_lock	26.4M	-26.4M	-	-	-	0
inode_lock	4.3M	-4.3M	-	-	-	0
slock	422.7K	+451.6K	-852.0K	-22.3K	-	0
ep.lock	1.0M	+906.4K	-1.3M	-649.7K	-	0
base.lock	451.3K	+337.2K	-277.6K	-510.9K	-	8
ehash.lock	868	+754	+466	+499	-2.6K	0

of incoming packets are received and processed on different CPU cores. In this case, the software Receive Flow Deliver forwards these non-local packets to the right CPU core to avoid further overhead from non-local connection problem. This reduces L3 cache miss rate by 6 percentage point and consequently improves throughput by 6.1% (from 261k to 277k).

For FDir_ATR, the local packet proportion rises to 76.5% and Receive Flow Deliver provides another performance improvement of 0.8% compared to FDir_ATR alone. Though the L3 cache miss rate when using FDir_ATR is close to RSS+RFD, FDir_ATR delivers a majority of packets to the right CPU core in NIC hardware and thus reduces software overhead on packet steering. At the same time, Receive Flow Deliver makes FDir_Perfect practical for accurately distributing incoming packets with a 100% local packet proportion, which gives us 1.8% reduction in L3 cache miss rate and 2.4% throughput improvement (from 293k to 300k) compared to only using FDir_ATR.

5. Related Work

Reducing Shared Resource Contentions In [14, 21], `inode` and `dentry` of VFS has been identified as the bottleneck for socket scalability and a more fine-grained lock scheme is proposed to mitigate inter-core shared resource contention. Megapipe [21] proposes a new channel design to bypass VFS completely. However, Megapipe loses compatibility with VFS abstraction. This forces legacy applications to modify their code and causes some system tools to misbehave, such as *netstat* and *lsof*. As described in Section 3.4, we adopt a more moderate design which retains the compatibility and remove locks associated with inode and dentry.

Locks in the listen socket is another hot spot identified by recent researches. Affinity-Accept [29], SO_REUSEPORT [23] and Megapipe [21] solve the single socket problem by cloning a listen socket copy for each CPU core. But these designs are still based on the global listen table. This is a potential scalability bottleneck Fastsocket has solved the problem completely with Local Listen Table.

Achieving Connection Locality Some researchs [34, 36, 38, 40–42] analyze and modify modern OSes, such as OpenSolaris, FreeBSD, Linux, Windows, etc., for high-performance TCP/IP network. In [38], two variations on a partitioned network-stack design for scalability in FreeBSD are described, and compared in evaluation with a conventional locked implementation. One variation protects partitioned data structures with locks that are global, but in practice uncontended. A second associates partitioned data structures whose instances may be accessed only by specific threads, providing synchronisation; Willman describes substantial overhead from context switching. Affinity-Accept [29] and Megapipe [21] achieve acceptable connection locality in the case of passive connection. Fastsocket goes further by realizing complete connection locality on both passive and active connections with Local Listen Table and Receive Flow Deliver.

Batching System Calls and zero-copy Communication FlexSC [32] and Megapipe [21] proposed system call batching to save the cost of context switch. It is possible to implement system call batching in Fastsocket and simulate the BSD socket API in the user library. Since we focus on the scalability problem in TCP stacks in this paper, integrating system call batching is left as future work. Zero-copy reduces data movement overheads and simplifies resource management [28]. POSIX OSes have been modified to support zero-copy through page remap- ping and copy-on-write [15]. Fastsocket can use zero-copy technologies in POSIX OSes [33].

Relaxing System Call Restrictions on Semantics Existing literatures [16, 21] have blamed the *lowest available file descriptor* in POSIX specification for poor scalability and relaxed the rule. However, there are applications implemented based on the rule. For example, the latest HAProxy assumes that file descriptor should never exceed the current connection number based on the file descriptor rule and uses file descriptor as the index of a connection array for connection management [1]. To achieve compatibility, we have not introduced these semantics changes in Fastsocket.

User-level Network Stacks Application-specific extensibility was one of the main motivations behind user-level networking. For example, application-specific customization of the networking stack in a user-level networking system was shown to be beneficial by enabling application-controlled flow control and feedback. There have been several user-level network stacks that try to accelerate the performance [17–20, 22]. However, they lack the full BSD socket API,

general and robust implementation of kernel network stack, as described in Section 1. Fastsocket, on the other hand, is built into the Linux kernel and is designed to be compatible with both existing applications and mature network stack components in the kernel. The compatibility achieved by Fastsocket makes it easier to deploy our proposal to production systems.

Library operating systems with hardware virtualization Library operating systems: Exokernels extend the end-to-end principle to resource management by implementing system abstractions via library operating systems linked in with applications [19]. Library operating systems often run as virtual machines. Hardware support for virtualization naturally separates control and execution functions, e.g., to build type-2 hypervisors [35], run virtual appliances [24]. IX [12] and Arrakis [30] uses hardware virtualization to separate the I/O dataplane from the control plane. IX uses a full Linux kernel as the control plane; provides three-way isolation between the control plane, networking stack, and application; and proposes a dataplane architecture that optimizes for both high throughput and low latency. Arrakis uses Barrelfish [11] as the control plane and includes support for IOMMUs and SR-IOV. As far as we know, IX and Arrakis use modified embedded TCP stack lwIP, which missed many advanced features as described in Section 1, which are very useful in practice.

Reserving Cores for Network IsoStack [31] aims to improve network system efficiency with a novel architecture that designates one CPU core to handle all network stack work to remove contentions, while all other CPU cores run applications. However, when adopting IsoStack in 10G and even 40G network, the dedicated single CPU core will be overloaded, especially in the CPU-intensive short-lived connection scenarios. Fastsocket shows that full partition of TCB management is a more efficient and feasible alternative to scale TCP stack.

6. Conclusion

This paper introduces Fastsocket, a new scalable TCP socket which successfully partitions shared socket tables including the listen table and established table. Fastsocket further implements PerCore Process Zone to avoid lock contentions and keep compatibility at the same time.

The key designs are 1) Local Listen Table which partitions the global listen table and guarantees passive connection locality, 2) Local Established Table which partitions the global established table and guarantee full partition on TCB management. 3) Receive Flow Deliver which guarantees active connection locality which coexists with passive connection locality, and 4) Fastsocket-aware VFS which avoids heavy lock contentions on sockets while keeping compatible with existing applications.

Nginx and HAProxy evaluation on Intel 24-core shows Fastsocket improves the performance by 267% and 621% respectively and retains full BSD socket API, which benefits existing socket applications and their developers as well. Moreover, all these modifications are not intrusive to the Linux kernel, which means Fastsocket can gracefully fit into the existing Linux kernel. Because of both high scalability and compatibility, Fastsocket has been deployed in Sina Weibo system to serve requests from all over the world. All Fastsocket source code is publicly available at https://github.com/fastos/fastsocket.

Acknowledgments

This research is carried out by Sina&Tsinghua&Intel's cooperation. We thank Intel, especially Mr. Xu Chen, for their hardware and expertise. And also thank Sixing Xiao for his early research work. This research is supported by Natural Science Foundation of China under Grant No. 61170050, National Science and Technology Major Project of China (2012ZX01039-004),The National High Technology Research and Development Program of China (2015AA011505).

References

[1] Haproxy. `http://haproxy.1wt.eu/`.

[2] http_load - multiprocessing http test client. `http://www.acme.com/software/http_load/`.

[3] Hypertext transfer protocol – http/1.0. `http://tools.ietf.org/html/rfc1945`.

[4] Lock statistics. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/lockstat.txt`.

[5] Tcp syn flooding attacks and common mitigations. `http://tools.ietf.org/html/rfc4987`.

[6] Rfs hardware acceleration. `http://lwn.net/Articles/406489/`, 2010.

[7] rfs: Receive flow steering. `http://lwn.net/Articles/381955/`, 2010.

[8] xps: Transmit packet steering. `http://lwn.net/Articles/412062/`, 2010.

[9] Intel 82599 10 gigabit ethernet controller datasheet. `http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html`, 2014.

[10] Intel i/o acceleration technology: Intel network adapters user guide. `http://web.mit.edu/cron/documentation/dell-server-admin/en/IntelNIC/ioat.htm`, 2014.

[11] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[12] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the*

11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, pages 49–65, Berkeley, CA, USA, 2014. USENIX Association.

[13] S. M. Bellovin. A look back at "security problems in the tcp/ip protocol suite". In *Proceedings of the 20th Annual Computer Security Applications Conference*, ACSAC '04, pages 229–249, Washington, DC, USA, 2004. IEEE Computer Society.

[14] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In R. H. Arpaci-Dusseau and B. Chen, editors, *OSDI*, pages 1–16. USENIX Association, 2010.

[15] H.-k. J. Chu. Zero-copy tcp in solaris. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 21–21, Berkeley, CA, USA, 1996. USENIX Association.

[16] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: designing scalable software for multicore processors. In M. Kaminsky and M. Dahlin, editors, *SOSP*, pages 1–17. ACM, 2013.

[17] S. Communications. Introduction to openonload: Building application transparency and protocol conformance into application acceleration middleware. `http://www.solarflare.com/content/userfiles/documents/solarflare_openonload_intropaper.pdf`, 2011.

[18] D. Ely, S. Savage, and D. Wetherall. Alpine: A user-level infrastructure for network protocol development. In *USITS*. USENIX, 2001.

[19] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceo, R. Hunt, T. Pinckney, and V. Inc. Fast and flexible application-level networking on exokernel systems. *ACM Transactions on Computer Systems*, 20:49–83, 2000.

[20] H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Deploying safe user-level network services with icTCP. In *OSDI*, pages 317–332. USENIX Association, 2004.

[21] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. Megapipe: A new programming interface for scalable network i/o. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 135–148, Berkeley, CA, USA, 2012. USENIX Association.

[22] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mtcp: a highly scalable user-level tcp stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, Apr. 2014. USENIX Association.

[23] M. Kerrisk. The so_reuseport socket option. `http://lwn.net/Articles/542629/`, 2013.

[24] I. Krsul, A. Ganguly, J. Zhang, J. A. B. Fortes, and R. J. Figueiredo. Vmplants: Providing and managing virtual machine execution environments for grid computing. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 7–, Washington, DC, USA, 2004. IEEE Computer Society.

[25] G. Loukas and G. Öke. Protection against denial of service attacks. *Comput. J.*, 53(7):1020–1037, Sept. 2010.

[26] lwIP community. lwip - a lightweight tcp/ip stack - summary. `http://savannah.nongnu.org/projects/lwip/`, 2012.

[27] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley. Performance issues in parallelized network protocols. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

[28] V. S. Pai, P. Druschel, and W. Zwaenepoel. Io-lite: A unified i/o buffering and caching system. *ACM Trans. Comput. Syst.*, 18(1):37–66, Feb. 2000.

[29] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In P. Felber, F. Bellosa, and H. Bos, editors, *EuroSys*, pages 337–350. ACM, 2012.

[30] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.

[31] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. Isostack: Highly efficient network processing on dedicated cores. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[32] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In R. H. Arpaci-Dusseau and B. Chen, editors, *OSDI*, pages 33–46. USENIX Association, 2010.

[33] T. Suzumura, M. Tatsubori, S. Trent, A. Tozawa, and T. Onodera. Highly scalable web applications with zero-copy data transfer. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 921–930, New York, NY, USA, 2009. ACM.

[34] S. Tripathi. Fireenginea new networking architecture for the solaris operating system. `http://www.scn.rain.com/~neighorn/PDF/FireEngine_WP.pdf`, 2004.

[35] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, May 2005.

[36] R. N. M. Watson. Introduction to multithreading and multiprocessing in the freebsd smpng network stack. `http://www.watson.org/~robert/freebsd/netperf/20051027-eurobsdcon2005-netperf.pdf`, 2005.

[37] D. F. Williamson, R. A. Parker, and J. S. Kendrick. The box plot: a simple visual method to interpret data. *Annals of internal medicine*, 110(11):916–921, 1989.

[38] P. Willmann, S. Rixner, and A. L. Cox. An evaluation of network stack parallelization strategies in modern operating systems. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 8–8, Berkeley, CA, USA, 2006. USENIX Association.

[39] S. Woo, E. Jeong, S. Park, J. Lee, S. Ihm, and K. Park. Comparison of caching strategies in modern cellular backhaul

networks. In H.-H. Chu, P. Huang, R. R. Choudhury, and F. Zhao, editors, *MobiSys*, pages 319–332. ACM, 2013.

[40] P. Xie, B. Wu, M. Liu, J. Harris, and C. Scheiman. Profiling the performance of tcp/ip on windows nt. *Computer Performance and Dependability Symposium, International*, 0:133, 2000.

[41] H. youb Kim and S. Rixner. Performance characterization of the freebsd network stack. `http://www.cs.rice.edu/CS/Architecture/docs/kim-tr05.pdf`, 2005.

[42] H. Zou, W. Wu, X.-H. Sun, P. DeMar, and M. Crawford. An evaluation of parallel optimization for opensolaris network stack. In *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*, pages 296–299, Oct 2010.