MegaPipe: A New Programming Interface for Scalable Network I/O

Sangjin Han[+], Scott Marshall[+], Byung-Gon Chun[*], and Sylvia Ratnasamy[+]

[+]*University of California, Berkeley* [*]*Yahoo! Research*

Abstract

We present MegaPipe, a new API for efficient, scalable network I/O for message-oriented workloads. The design of MegaPipe centers around the abstraction of a *channel* – a per-core, bidirectional pipe between the kernel and user space, used to exchange both I/O requests and event notifications. On top of the channel abstraction, we introduce three key concepts of MegaPipe: partitioning, lightweight socket (lwsocket), and batching.

We implement MegaPipe in Linux and adapt memcached and nginx. Our results show that, by embracing a clean-slate design approach, MegaPipe is able to exploit new opportunities for improved performance and ease of programmability. In microbenchmarks on an 8-core server with 64 B messages, MegaPipe outperforms baseline Linux between 29% (for long connections) and 582% (for short connections). MegaPipe improves the performance of a modified version of memcached between 15% and 320%. For a workload based on real-world HTTP traces, MegaPipe boosts the throughput of nginx by 75%.

1 Introduction

Existing network APIs on multi-core systems have difficulties scaling to high connection rates and are inefficient for "message-oriented" workloads, by which we mean workloads with short connections[1] *and/or* small messages. Such message-oriented workloads include HTTP, RPC, key-value stores with small objects (e.g., RAMCloud [31]), etc. Several research efforts have addressed aspects of these performance problems, proposing new techniques that offer valuable performance improvements. However, they all innovate within the confines of the traditional socket-based networking APIs, by either *i)* modifying the internal implementation but leaving the APIs untouched [20, 33, 35], or *ii)* adding new APIs to complement the existing APIs [1, 8, 10, 16, 29]. While these approaches have the benefit of maintaining backward compatibility for existing applications, the need to maintain the *generality* of the existing API – e.g., its reliance on file descriptors, support for blocking and nonblocking communication, asynchronous I/O, event polling, and so forth – limits the extent to which it can be optimized for performance. In contrast, a clean-slate redesign offers the opportunity to present an API that is specialized for high performance network I/O.

An ideal network API must offer not only high performance but also a simple and intuitive programming abstraction. In modern network servers, achieving high performance requires efficient support for *concurrent I/O* so as to enable scaling to large numbers of connections per thread, multiple cores, etc. The original socket API was not designed to support such concurrency. Consequently, a number of new programming abstractions (e.g., `epoll`, `kqueue`, etc.) have been introduced to support concurrent operation without overhauling the socket API. Thus, even though the basic socket API is simple and easy to use, programmers face the unavoidable and tedious burden of layering several abstractions for the sake of concurrency. Once again, a clean-slate design of network APIs offers the opportunity to design a network API from the ground up with support for concurrent I/O.

Given the central role of networking in modern applications, we posit that it is worthwhile to explore the benefits of a clean-slate design of network APIs aimed at achieving both high performance and ease of programming. In this paper we present MegaPipe, a new API for efficient, scalable network I/O. The core abstraction MegaPipe introduces is that of a *channel* – a per-core, bi-directional pipe between the kernel and user space that is used to exchange both asynchronous I/O requests *and* completion notifications. Using channels, MegaPipe achieves high performance through three design contributions under the roof of a single unified abstraction:

Partitioned listening sockets: Instead of a single listening socket shared across cores, MegaPipe allows applications to clone a listening socket and partition its associated queue across cores. Such partitioning improves performance with multiple cores while giving applications control over their use of parallelism.

Lightweight sockets: Sockets are represented by file descriptors and hence inherit some unnecessary file-related overheads. MegaPipe instead introduces lwsocket, a lightweight socket abstraction that is not wrapped in file-related data structures and thus is free from system-wide synchronization.

[1] We use "short connection" to refer to a connection with a small number of messages exchanged; this is not a reference to the absolute time duration of the connection.

System Call Batching: MegaPipe amortizes system call overheads by batching asynchronous I/O requests and completion notifications within a channel.

We implemented MegaPipe in Linux and adapted two popular applications – memcached [3] and the nginx [37] – to use MegaPipe. In our microbenchmark tests on an 8-core server with 64 B messages, we show that MegaPipe outperforms the baseline Linux networking stack between 29% (for long connections) and 582% (for short connections). MegaPipe improves the performance of a modified version of memcached between 15% and 320%. For a workload based on real-world HTTP traffic traces, MegaPipe improves the performance of nginx by 75%.

The rest of the paper is organized as follows. We expand on the limitations of existing network stacks in §2, then present the design and implementation of MegaPipe in §3 and §4, respectively. We evaluate MegaPipe with microbenchmarks and macrobenchmarks in §5, and review related work in §6.

2 Motivation

Bulk transfer network I/O workloads are known to be inexpensive on modern commodity servers; one can easily saturate a 10 Gigabit (10G) link utilizing only a single CPU core. In contrast, we show that message-oriented network I/O workloads are very CPU-intensive and may significantly degrade throughput. In this section, we discuss limitations of the current BSD socket API (§2.1) and then quantify the performance with message-oriented workloads with a series of RPC-like microbenchmark experiments (§2.2).

2.1 Performance Limitations

In what follows, we discuss known sources of inefficiency in the BSD socket API. Some of these inefficiencies are general, in that they occur even in the case of a single core, while others manifest only when scaling to multiple cores – we highlight this distinction in our discussion.

Contention on Accept Queue (multi-core): As explained in previous work [20, 33], a single listening socket (with its `accept()` backlog queue and exclusive lock) forces CPU cores to serialize queue access requests; this hotspot negatively impacts the performance of both producers (kernel threads) enqueueing new connections and consumers (application threads) accepting new connections. It also causes CPU cache contention on the shared listening socket.

Lack of Connection Affinity (multi-core): In Linux, incoming packets are distributed across CPU cores on a flow basis (hash over the 5-tuple), either by hardware (RSS [5]) or software (RPS [24]); all receive-side processing for the flow is done on a core. On the other hand, the transmit-side processing happens on the core at which the application thread for the flow resides. Because of the serialization in the listening socket, an application thread calling `accept()` may accept a new connection that came through a remote core; RX/TX processing for the flow occurs on two different cores, causing expensive cache bouncing on the TCP control block (TCB) between those cores [33]. While the per-flow redirection mechanism [7] in NICs eventually resolves this core disparity, short connections cannot benefit since the mechanism is based on packet sampling.

File Descriptors (single/multi-core): The POSIX standard requires that a newly allocated file descriptor be the lowest integer not currently used by the process [6]. Finding 'the first hole' in a file table is an expensive operation, particularly when the application maintains many connections. Even worse, the search process uses an explicit per-process lock (as files are shared within the process), limiting the scalability of multi-threaded applications. In our `socket()` microbenchmark on an 8-core server, the cost of allocating a single FD is roughly 16% greater when there are 1,000 existing sockets as compared to when there are no existing sockets.

VFS (multi-core): In UNIX-like operating systems, network sockets are abstracted in the same way as other file types in the kernel; the Virtual File System (VFS) [27] associates each socket with corresponding file instance, inode, and dentry data structures. For message-oriented workloads with short connections, where sockets are frequently opened as new connections arrive, servers quickly become overloaded since those globally visible objects cause system-wide synchronization cost [20]. In our microbenchmark, the VFS overhead for socket allocation on eight cores was 4.2 times higher than the single-core case.

System Calls (single-core): Previous work has shown that system calls are expensive and negatively impact performance, both directly (mode switching) and indirectly (cache pollution) [35]. This performance overhead is exacerbated for message-oriented workloads with small messages that result in a large number of I/O operations.

In parallel with our work, the Affinity-Accept project [33] has recently identified and solved the first two issues, both of which are caused by the shared listening socket (for complete details, please refer to the paper). We discuss our approach (partitioning) and its differences in §3.4.1. To address other issues, we introduce the concept of lwsocket (§3.4.2, for FD and VFS overhead) and batching (§3.4.3, for system call overhead).

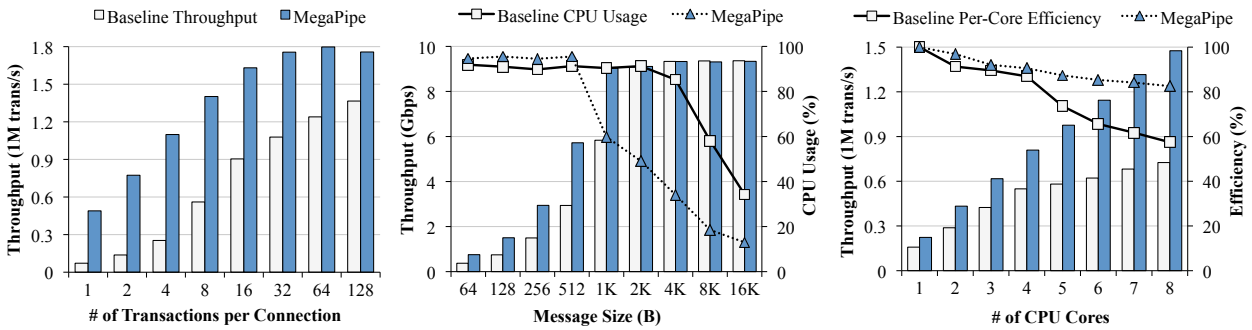

Figure 1: (a) the negative impact of connection lifespan (with 64 B messages on eight cores), (b) message size (with ten transactions per connection on eight cores), and (c) increasing number of cores (with 64 B messages and ten transactions per connection).

2.2 Performance of Message-Oriented Workloads

While it would be ideal to separate the aforementioned inefficiencies and quantify the cost of each, tight coupling in semantics between those issues and complex dynamics of synchronization/cache make it challenging to isolate individual costs.

Rather, we quantify their *compound* performance impact with a series of microbenchmarks in this work. As we noted, the inefficiencies manifest themselves primarily in workloads that involve *short connections* or *small-sized messages*, particularly with increasing numbers of CPU cores. Our microbenchmark tests thus focus on these problematic scenarios.

Experimental Setup: For our tests, we wrote a pair of client and server microbenchmark tools that emulate RPC-like workloads. The client initiates a TCP connection, exchanges multiple request and response messages with the server and then closes the connection.[2] We refer to a single request-response exchange as a *transaction*. Default parameters are 64 B per message and 10 transactions per connection, unless otherwise stated. Each client maintains 256 concurrent connections, and we confirmed that the client is never the bottleneck. The server creates a single listening socket shared by eight threads, with each thread pinned to one CPU core. Each event-driven thread is implemented with `epoll` [8] and the non-blocking socket API.

Although synthetic, this workload lets us focus on the low-level details of network I/O overhead without interference from application-specific logic. We use a single server and three client machines, connected through a dedicated 10G Ethernet switch. All test systems use the Linux 3.1.3 kernel and ixgbe 3.8.21 10G Ethernet device driver [2] (with interrupt coalescing turned on). Each machine has a dual-port Intel 82599 10G NIC, 12 GB of DRAM, and two Intel Xeon X5560 processors, each of which has four 2.80 GHz cores. We enabled the multi-queue feature of the NICs with RSS [5] and FlowDirector [7], and assigned each RX/TX queue to one CPU core.

In this section, we discuss the result of the experiments Figure 1 labeled as "Baseline." For comparison, we also include the results with our new API, labeled as "MegaPipe," from the same experiments.

(a) Performance with Short Connections: TCP connection establishment involves a series of time-consuming steps: the 3-way handshake, socket allocation, and interaction with the user-space application. For workloads with short connections, the costs of connection establishment are not amortized by sufficient data transfer and hence this workload serves to highlight the overhead due to costly connection establishment.

We show how connection lifespan affects the throughput by varying the number of transactions per connection in Figure 1(a), measured with eight CPU cores. Total throughput is significantly lower with relatively few (1–8) transactions per connection. The cost of connection establishment eventually becomes insignificant for 128+ transactions per connection, and we observe that throughput in single-transaction connections is roughly 19 times lower than that of long connections!

(b) Performance with Small Messages: Small messages result in greater relative network I/O overhead in comparison to larger messages. In fact, the per-message overhead remains roughly constant and thus, independent of message size; in comparison with a 64 B message, a 1 KiB message adds only about 2% overhead due to the copying between user and kernel on our system, despite the large size difference.

To measure this effect, we perform a second microbenchmark with response sizes varying from 64 B to 64 KiB (varying the request size in lieu of or in addition to the response size had almost the same effects). Figure 1(b) shows the measured throughput (in Gbps) and CPU usage for various message sizes. It is clear that connections with

[2] In this experiment, we closed connections with RST, to avoid exhaustion of client ports caused by lingering TIME_WAIT connections.

small-sized messages adversely affect the throughput. For small messages ($\leq$ 1 KiB) the server does not even saturate the 10G link. For medium-sized messages (2–4 KiB), the CPU utilization is extremely high, leaving few CPU cycles for further application processing.

(c) Performance Scaling with Multiple Cores: Ideally, throughput for a CPU-intensive system should scale linearly with CPU cores. In reality, throughput is limited by shared hardware (e.g., cache, memory buses) and/or software implementation (e.g., cache locality, serialization). In Figure 1(c), we plot the throughput for increasing numbers of CPU cores. To constrain the number of cores, we adjust the number of server threads and RX/TX queues of the NIC. The lines labeled "Efficiency" represent the measured per-core throughput, normalized to the case of perfect scaling, where *N* cores yield a speedup of *N*.

We see that throughput scales relatively well for up to four cores – the likely reason being that, since each processor has four cores, expensive off-chip communication does not take place up to this point. Beyond four cores, the marginal performance gain with each additional core quickly diminishes, and with eight cores, speedup is only 4.6. Furthermore, it is clear from the growth trend that speedup would not increase much in the presence of additional cores. Finally, it is worth noting that the observed scaling behavior of Linux highly depends on connection duration, further confirming the results in Figure 1(a). With only one transaction per connection (instead of the default 10 used in this experiment), the speedup with eight cores was only 1.3, while longer connections of 128 transactions yielded a speedup of 6.7.

3 MegaPipe Design

MegaPipe is a new programming interface for high-performance network I/O that addresses the inefficiencies highlighted in the previous section and provides an easy and intuitive approach to programming high concurrency network servers. In this section, we present the design goals, approach, and contributions of MegaPipe.

3.1 Scope and Design Goals

MegaPipe aims to accelerate the performance of message-oriented workloads, where connections are short and/or message sizes are small. Some possible approaches to this problem would be to extend the BSD Socket API or to improve its internal implementation. It is hard to achieve optimal performance with these approaches, as many optimization opportunities can be limited by the legacy abstractions. For instance: *i)* sockets represented as files inherit the overheads of files in the kernel; *ii)* it is difficult to aggregate BSD socket operations from concurrent connections to amortize system call overheads. We leave optimizing the message-oriented workloads with those dirty-slate (minimally disruptive to existing API semantics and legacy applications) alternatives as an open problem. Instead, we take a clean-slate approach in this work by designing a new API from scratch.

We design MegaPipe to be conceptually simple, self-contained, and applicable to existing event-driven server applications with moderate efforts. The MegaPipe API provides a unified interface for various I/O types, such as TCP connections, UNIX domain sockets, pipes, and disk files, based on the completion notification model (§3.2) We particularly focus on the performance of network I/O in this paper. We introduce three key design concepts of MegaPipe for high-performance network I/O: partitioning (§3.4.1), lwsocket (§3.4.2), and batching (§3.4.3), for reduced per-message overheads and near-linear multi-core scalability.

3.2 Completion Notification Model

The current best practice for event-driven server programming is based on the readiness model. Applications poll the readiness of interested sockets with `select/poll/epoll` and issue non-blocking I/O commands on the those sockets. The alternative is the completion notification model. In this model, applications issue asynchronous I/O commands, and the kernel notifies the applications when the commands are complete. This model has rarely been used for network servers in practice, though, mainly because of the lack of socket-specific operations such as `accept/connect/shutdown` (e.g., POSIX AIO [6]) or poor mechanisms for notification delivery (e.g., SIGIO signals).

MegaPipe adopts the completion notification model over the readiness model for three reasons. First, it allows transparent batching of I/O commands and their notifications. Batching of non-blocking I/O commands in the readiness model is very difficult without the explicit assistance from applications. Second, it is compatible with not only sockets but also disk files, allowing a unified interface for any type of I/O. Lastly, it greatly simplifies the complexity of I/O multiplexing. Since the kernel controls the rate of I/O with completion events, applications can blindly issue I/O operations without tracking the readiness of sockets.

3.3 Architectural Overview

MegaPipe involves both a user-space library and Linux kernel modifications. Figure 2 illustrates the architecture and highlights key abstractions of the MegaPipe design. The left side of the figure shows how a multi-threaded application interacts with the kernel via MegaPipe *channels*. With MegaPipe, an application thread running on each core opens a separate channel for communication between the kernel and user-space. The application thread

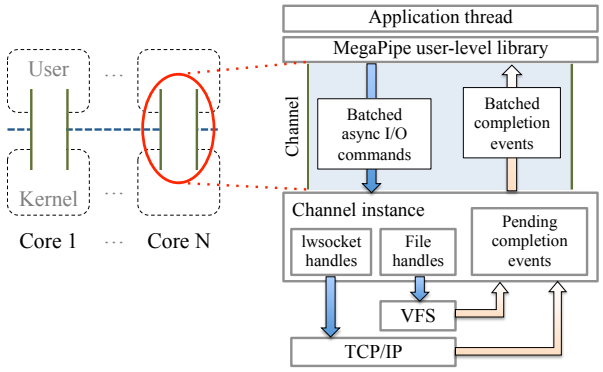

Figure 2: MegaPipe architecture

registers a *handle* (socket or other file type) to the channel, and each channel multiplexes its own set of handles for their asynchronous I/O requests and completion notification events.

When a listening socket is registered, MegaPipe internally spawns an independent accept queue for the channel, which is responsible for incoming connections to the core. In this way, the listening socket is not shared by all threads, but *partitioned* (§3.4.1) to avoid serialization and remote cache access.

A handle can be either a regular file descriptor or a lightweight socket, *lwsocket* (§3.4.2). lwsocket provides a direct shortcut to the TCB in the kernel, to avoid the VFS overhead of traditional sockets; thus lwsockets are only visible within the associated channel.

Each channel is composed of two message streams: a request stream and a completion stream. User-level applications issue asynchronous I/O requests to the kernel via the request stream. Once the asynchronous I/O request is done, the completion notification of the request is delivered to user-space via the completion stream. This process is done in a *batched* (§3.4.3) manner, to minimize the context switch between user and kernel. The MegaPipe user-level library is fully responsible for transparent batching; MegaPipe does not need to be aware of batching.

3.4 Design Components

3.4.1 Listening Socket Partitioning

As discussed in §2.1, the shared listening socket causes two issues in the multi-core context: *i)* contention on the accept queue and *ii)* cache bouncing between RX and TX cores for a flow. Affinity-Accept [33] proposes two key ideas to solve these issues. First, a listening socket has per-core accept queues instead of the shared one. Second, application threads that call `accept()` prioritize their local accept queue. In this way, connection establishment becomes completely parallelizable and independent, and all the connection establishment, data transfer, and application logic for a flow are contained in the same core.

In MegaPipe, we achieve essentially the same goals but with a more controlled approach. When an application thread associates a listening socket to a channel, MegaPipe spawns a separate listening socket. The new listening socket has its own accept queue which is only responsible for connections established on a particular subset of cores that are explicitly specified by an optional `cpu_mask` parameter.[3] After a shared listening socket is registered to MegaPipe channels with disjoint `cpu_mask` parameters, all channels (and thus cores) have completely partitioned backlog queues. Upon receipt of an incoming TCP handshaking packet, which is distributed across cores either by RSS [5] or RPS [24], the kernel finds a "local" accept queue among the partitioned set, whose `cpu_mask` includes the current core. On the application side, an application thread accepts pending connections from its local queue. In this way, cores no longer contend for the shared accept queue, and connection establishment is vertically partitioned (from the TCP/IP stack up to the application layer).

We briefly discuss the main difference between our technique and that of Affinity-Accept. Our technique requires user-level applications to partition a listening socket explicitly, rather than transparently. The downside is that legacy applications do not benefit. However, explicit partitioning provides more flexibility for user applications (e.g., to forgo partitioning for single-thread applications, to establish one accept queue for each physical core in SMT systems, etc.) Our approach follows the design philosophy of the Corey operating system, in a way that "applications should control sharing" [19].

Partitioning of a listening socket may cause potential load imbalance between cores [33]. Affinity-Accept solves two cases of load imbalance. For a short-term load imbalance, a non-busy core running `accept()` may steal a connection from the remote accept queue on a busy CPU core. For a long-term load imbalance, the flow group migration mechanism lets the NIC to distribute more flows to non-busy cores. While the current implementation of MegaPipe does not support load balancing of incoming connections between cores, the techniques made in Affinity-Accept are complementary to MegaPipe. We leave the implementation and evaluation of connection load balancing as future work.

3.4.2 lwsocket: Lightweight Socket

`accept()`ing an established connection is an expensive process in the context of the VFS layer. In Unix-like operating systems, many different types of open files (disk files, sockets, pipes, devices, etc.) are identified by a *file*

[3] MegaPipe currently does not support runtime reconfiguration of `cpu_mask` after it is initially set, but we believe that this is easy to add.

descriptor. A file descriptor is an integer identifier used as an indirect reference to an opened *file instance*, which maintains the status (e.g., access mode, offset, and flags such as `O_DIRECT` and `O_SYNC`) of the opened file. Multiple file instances may point to the same *inode*, which represents a unique, permanent file object. An inode points to an actual type-specific kernel object, such as TCB.

These layers of abstraction offer clear advantages. The kernel can seamlessly support various file systems and file types, while retaining a unified interface (e.g., `read()` and `write()`) to user-level applications. The CPU overhead that comes with the abstraction is tolerable for regular disk files, as file I/O is typically bound by low disk bandwidth or high seek latency. For network sockets, however, we claim that these layers of abstraction could be overkill for the following reasons:

(1) Sockets are rarely shared. For disk files, it is common that multiple processes share the same open file or independently open the same permanent file. The layer of indirection that file objects offer between the file table and inodes is useful in such cases. In contrast, since network sockets are rarely shared by multiple processes (HTTP socket redirected to a CGI process is such an exception) and not opened multiple times, this indirection is typically unnecessary.

(2) Sockets are ephemeral. Unlike permanent disk-backed files, the lifetime of network sockets ends when they are closed. Every time a new connection is established or torn down, its FD, file instance, inode, and dentry are newly allocated and freed. In contrast to disk files whose inode and dentry objects are cached [27], socket inode and dentry cannot benefit from caching since sockets are ephemeral. The cost of frequent (de)allocation of those objects is exacerbated on multi-core systems since the kernel maintains the inode and dentry as globally visible data structures [20].

To address the above issues, we propose lightweight sockets – *lwsocket*. Unlike regular files, a lwsocket is identified by an arbitrary integer within the channel, not the lowest possible integer within the process. The lwsocket is a common-case optimization for network connections; it does not create a corresponding file instance, inode, or dentry, but provides a straight shortcut to the TCB in the kernel. A lwsocket is only locally visible within the associated MegaPipe channel, which avoids global synchronization between cores.

In MegaPipe, applications can choose whether to fetch a new connection as a regular socket or as a lwsocket. Since a lwsocket is associated with a specific channel, one cannot use it with other channels or for general system calls, such as `sendmsg()`. In cases where applications need the full generality of file descriptors, MegaPipe provides a fall-back API function to convert a lwsocket into a regular file descriptor.

3.4.3 System Call Batching

Recent research efforts report that system calls are expensive not only due to the cost of mode switching, but also because of the negative effect on cache locality in both user and kernel space [35]. To amortize system call costs, MegaPipe batches multiple I/O requests and their completion notifications into a single system call. The key observation here is that batching can exploit connection-level parallelism, extracting multiple independent requests and notifications from concurrent connections.

Batching is transparently done by the MegaPipe user-level library for both directions user $\rightarrow$ kernel and kernel $\rightarrow$ user. Application programmers need not be aware of batching. Instead, application threads issue one request at a time, and the user-level library accumulates them. When *i)* the number of accumulated requests reaches the batching threshold, *ii)* there are not any more pending completion events from the kernel, or *iii)* the application explicitly asks to flush, then the collected requests are flushed to the kernel in a batch through the channel. Similarly, application threads dispatch a completion notification from the user-level library one by one. When the user-level library has no more completion notifications to feed the application thread, it fetches multiple pending notifications from kernel in a batch. We set the default batching threshold to 32 (adjustable), as we found that the marginal performance gain beyond that point is negligible.

3.5 API

The MegaPipe user-level library provides a set of API functions to hide the complexity of batching and the internal implementation details. Table 1 presents a partial list of MegaPipe API functions. Due to lack of space, we highlight some interesting aspects of some functions rather than enumerating all of them.

The application associates a handle (either a regular file descriptor or a lwsocket) with the specified channel with `mp_register()`. All further I/O commands and completion notifications for the registered handle are done through only the associated channel. A cookie, an opaque pointer for developer use, is also passed to the kernel with handle registration. This cookie is attached in the completion events for the handle, so the application can easily identify which handle fired each event. The application calls `mp_unregister()` to end the membership. Once unregistered, the application can continue to use the regular FD with general system calls. In contrast, lwsockets are immediately deallocated from the kernel memory.

When a listening TCP socket is registered with the

Function	Parameters	Description
`mp_create()`		Create a new MegaPipe channel instance.
`mp_register()`	channel, fd, cookie, cpu_mask	Create a MegaPipe handle for the specified file descriptor (either regular or lightweight) in the given channel. If a given file descriptor is a listening socket, an optional CPU mask parameter can be used to designate the set of CPU cores which will respond to incoming connections for that handle.
`mp_unregister()`	handle	Remove the target handle from the channel. All pending completion notifications for the handle are canceled.
`mp_accept()`	handle, count, is_lwsocket	Accept one or more new connections from a given listening handle asynchronously. The application specifies whether to accept a connection as a regular socket or a lwsocket. The completion event will report a new FD/lwsocket and the number of pending connections in the accept queue.
`mp_read()` `mp_write()`	handle, buf, size	Issue an asynchronous I/O request. The completion event will report the number of bytes actually read/written.
`mp_disconnect()`	handle	Close a connection in a similar way with `shutdown()`. It does not deallocate or unregister the handle.
`mp_dispatch()`	channel, timeout	Retrieve a single completion notification for the given channel. If there is no pending notification event, the call blocks until the specified timer expires.

Table 1: MegaPipe API functions (not exhaustive).

`cpu_mask` parameter, MegaPipe internally spawns an accept queue for incoming connections on the specified set of CPU cores. The original listening socket (now responsible for the remaining CPU cores) can be registered to other MegaPipe channels with a disjoint set of cores – so each thread can have a completely partitioned view of the listening socket.

`mp_read()` and `mp_write()` issue asynchronous I/O commands. The application should not use the provided buffer for any other purpose until the completion event, as the ownership of the buffer has been delegated to the kernel, like in other asynchronous I/O APIs. The completion notification is fired when the I/O is actually completed, i.e., all data has been copied from the receive queue for read or copied to the send queue for write. In adapting nginx and memcached, we found that vectored I/O operations (multiple buffers for a single I/O operation) are helpful for optimal performance. For example, the unmodified version of nginx invokes the `writev()` system call to transmit separate buffers for a HTTP header and body at once. MegaPipe supports the counterpart, `mp_writev()`, to avoid issuing multiple `mp_write()` calls or aggregating scattered buffers into one contiguous buffer.

`mp_dispatch()` returns one completion event as a `struct mp_event`. This data structure contains: *i)* a completed command type (e.g., read/write/accept/etc.), *ii)* a cookie, *iii)* a result field that indicates success or failure (such as broken pipe or connection reset) with the corresponding `errno` value, and *iv)* a `union` of command-specific return values.

Listing 1 presents simplified pseudocode of a ping-pong server to illustrate how applications use MegaPipe. An application thread initially creates a MegaPipe channel and registers a listening socket (`listen_sd` in this example) with `cpu_mask` 0x01 (first bit is set) which means that the handle is only interested in new connections established on the first core (core 0). The application then invokes `mp_accept()` and is ready to accept new connections. The body of the event loop is fairly simple; given an event, the server performs any appropriate tasks (barely anything in this ping-pong example) and then fires new I/O operations.

```
ch = mp_create()
handle = mp_register(ch, listen_sd, mask=0x01)
mp_accept(handle)

while true:
  ev = mp_dispatch(ch)
  conn = ev.cookie
  if ev.cmd == ACCEPT:
    mp_accept(conn.handle)
    conn = new Connection()
    conn.handle = mp_register(ch, ev.fd,
        cookie=conn)
    mp_read(conn.handle, conn.buf, READSIZE)
  elif ev.cmd == READ:
    mp_write(conn.handle, conn.buf, ev.size)
  elif ev.cmd == WRITE:
    mp_read(conn.handle, conn.buf, READSIZE)
  elif ev.cmd == DISCONNECT:
    mp_unregister(ch, conn.handle)
    delete conn
```

Listing 1: Pseudocode for ping-pong server event loop

3.6 Discussion: Thread-Based Servers

The current MegaPipe design naturally fits event-driven servers based on callback or event-loop mechanisms [32, 40]. We mostly focus on event-driven servers in this work. On the other hand, MegaPipe is also applicable to thread-based servers, by having one channel for each thread, thus each connection. In this case the application cannot take advantage of batching (§3.4.3), since batching exploits the parallelism of independent connections that are multiplexed through a channel. However, the application still can benefit from partitioning (§3.4.1) and lwsocket

(§3.4.2) for better scalability on multi-core servers.

There is an interesting spectrum between pure event-driven servers and pure thread-based servers. Some frameworks expose thread-like environments to user applications to retain the advantages of thread-based architectures, while looking like event-driven servers to the kernel to avoid the overhead of threading. Such functionality is implemented in various ways: lightweight user-level threading [23, 39], closures or coroutines [4, 18, 28], and language runtime [14]. Those frameworks intercept I/O calls issued by user threads to keep the kernel thread from blocking, and manage the outstanding I/O requests with polling mechanisms, such as `epoll`. These frameworks can leverage MegaPipe for higher network I/O performance without requiring modifications to applications themselves. We leave the evaluation of effectiveness of MegaPipe for these frameworks as future work.

4 Implementation

We begin this section with how we implemented MegaPipe in the Linux kernel and the associated user-level library. To verify the applicability of MegaPipe, we show how we adapted two applications (memcached and nginx) to benefit from MegaPipe.

4.1 MegaPipe API Implementation

As briefly described in §3.3, MegaPipe consists of two parts: the kernel module and the user-level library. In this section, we denote them by MP-K and MP-L, respectively, for clear distinction between the two.

Kernel Implementation: MP-K interacts with MP-L through a special device, `/dev/megapipe`. MP-L opens this file to create a channel, and invokes `ioctl()` system calls on the file to issue I/O requests and dispatch completion notifications for that channel.

MP-K maintains a set of handles for both regular FDs and lwsockets in a red-black tree[4] for each channel. Unlike a per-process file table, each channel is only accessed by one thread, avoiding data sharing between threads (thus cores). MP-K identifies a handle by an integer unique to the owning channel. For regular FDs, the existing integer value is used as an identifier, but for lwsockets, an integer of 2^{30} or higher value is issued to distinguish lwsockets from regular FDs. This range is used since it is unlikely to conflict with regular FD numbers, as the POSIX standard allocates the lowest unused integer for FDs [6].

MP-K currently supports the following file types: sockets, pipes, FIFOs, signals (via `signalfd`), and timers (via `timerfd`). MP-K handles asynchronous I/O requests differently depending on the file type. For sockets (such as TCP, UDP, and UNIX domain), MegaPipe utilizes the native callback interface, which fires upon state changes, supported by kernel sockets for optimal performance. For other file types, MP-K internally emulates asynchronous I/O with `epoll` and non-blocking VFS operations within kernel. MP-K currently does not support disk files, since the Linux file system does not natively support asynchronous or non-blocking disk I/O, unlike other modern operating systems. To work around this issue, we plan to adopt a lightweight technique presented in FlexSC [35] to emulate asynchronous I/O. When a disk I/O operation is about to block, MP-K can spawn a new thread on demand while the current thread continues.

Upon receiving batched I/O commands from MP-L through a channel, MP-K first examines if each request can be processed immediately (e.g., there is pending data in the TCP receive queue, or there is free space in the TCP send queue). If so, MP-K processes the request and issues a completion notification immediately, without incurring the callback registration or `epoll` overhead. This idea of opportunistic shortcut is adopted from LAIO [22], where the authors claim that the 73–86% of I/O operations are readily available. For I/O commands that are not readily available, MP-K needs some bookkeeping; it registers a callback to the socket or declares an epoll interest for other file types. When MP-K is notified that the I/O operation has become ready, it processes the operation.

MP-K enqueues I/O completion notifications in the per-channel event queue. Those notifications are dispatched in a batch upon the request of MP-L. Each handle maintains a linked list to its pending notification events, so that they can be easily removed when the handle is unregistered (and thus not of interest anymore).

We implemented MP-K in the Linux 3.1.3 kernel with 2,200 lines of code in total. The majority was implemented as a Linux kernel module, such that the module can be used for other Linux kernel versions as well. However, we did have to make three minor modifications (about 400 lines of code of the 2,200) to the Linux kernel itself, due to the following issues: *i)* we modified `epoll` to expose its API to not only user space but also to MP-K; *ii)* we modified the Linux kernel to allow multiple sockets (partitioned) to listen on the same address/port concurrently, which traditionally is not allowed; and *iii)* we also enhanced the socket lookup process for incoming TCP handshake packets to consider `cpu_mask` when choosing a destination listening socket among a partitioned set.

User-Level Library: MP-L is essentially a simple wrapper of the kernel module, and it is written in about 400

[4] It was mainly for ease of implementation, as Linux provides the template of red-black trees. We have not yet evaluated alternatives, such as a hash table, which supports $O(1)$ lookup rather than $O(logN)$.

Application	Total	Changed
memcached	9442	602 (6.4%)
nginx	86774	447 (0.5%)

Table 2: Lines of code for application adaptations

lines of code. MP-L performs two main roles: *i)* it transparently provides batching for asynchronous I/O requests and their completion notifications, *ii)* it performs communication with MP-K via the `ioctl()` system call.

The current implementation uses copying to transfer commands (24 B for each) and notifications (40 B for each) between MP-L and MP-K. This copy overhead, roughly 3–5% of total CPU cycles (depending on workloads) in our evaluation, can be eliminated with virtual memory mapping for the command/notification queues, as introduced in Mach Port [11]. We leave the implementation and evaluation of this idea as future work.

4.2 Application Integration

We adapted two popular event-driven servers, memcached 1.4.13 [3] (an in-memory key-value store) and nginx 1.0.15 [37] (a web server), to verify the applicability of MegaPipe. As quantitatively indicated in Table 2, the code changes required to use MegaPipe were manageable, on the order of hundreds of lines of code. However, these two applications presented different levels of effort during the adaptation process. We briefly introduce our experiences here, and show the performance benefits in Section 5.

memcached: memcached uses the libevent [30] framework which is based on the readiness model (e.g., `epoll` on Linux). The server consists of a main thread and a collection of worker threads. The main thread accepts new client connections and distributes them among the worker threads. The worker threads run event loops which dispatch events for client connections.

Modifying memcached to use MegaPipe in place of libevent involved three steps[5]:

(1) Decoupling from libevent: We began by removing libevent-specific data structures from memcached. We also made the drop-in replacement of `mp_dispatch()` for the libevent event dispatch loop.

(2) Parallelizing accept: Rather than having a single thread that accepts all new connections, we modified worker threads to accept connections in parallel by partitioning the shared listening socket.

(3) State machine adjustment: Finally, we replaced calls to `read()` with `mp_read()` and calls to `sendmsg()` with `mp_writev()`. Due to the semantic gap between the readiness model and the completion notification model, each state of the memcached state machine that invokes a MegaPipe function was split into two states: actions prior to a MegaPipe function call, and actions that follow the MegaPipe function call and depend on its result. We believe this additional overhead could be eliminated if memcached did not have the strong assumption of the readiness model.

nginx: Compared to memcached, nginx modifications were much more straightforward due to three reasons: *i)* the custom event-driven I/O of nginx does not use an external I/O framework that has a strong assumption of the readiness model, such as libevent [30]; *ii)* nginx was designed to support not only the readiness model (by default with `epoll` in Linux), but also the completion notification model (for POSIX AIO [6] and signal-based AIO), which nicely fits with MegaPipe; and *iii)* all worker processes already accept new connections in parallel, but from the shared listening socket.

nginx has an extensible *event module* architecture, which enables easy replacement for its underlying event-driven mechanisms. Under this architecture, we implemented a MegaPipe event module and registered `mp_read()` and `mp_writev()` as the actual I/O functions. We also adapted the worker threads to accept new connections from the partitioned listening socket.

5 Evaluation

We evaluated the performance gains yielded by MegaPipe both through a collection of microbenchmarks, akin to those presented in §2.2, and a collection of application-level macrobenchmarks. Unless otherwise noted, all benchmarks were completed with the same experimental setup (same software versions and hardware platforms as described in §2.2.

5.1 Microbenchmarks

The purpose of the microbenchmark results is three-fold. First, utilization of the same benchmark strategy as in §2 allows for direct evaluation of the low-level limitations we previously highlighted. Figure 1 shows the performance of MegaPipe measured for the same experiments. Second, these microbenchmarks give us the opportunity to measure an upper-bound on performance, as the minimal benchmark program effectively rules out any complex effects from application-specific behaviors. Third, microbenchmarks allow us to illuminate the performance contributions of each of MegaPipe's individual design components.

We begin with the impact of MegaPipe on multi-core scalability. Figure 3 provides a side-by-side comparison of parallel speedup (compared to the single core case of

[5] In addition, we pinned each worker thread to a CPU core for the MegaPipe adaptation, which is considered a best practice and is necessary for MegaPipe. We made the same modification to stock memcached for a fair comparison.

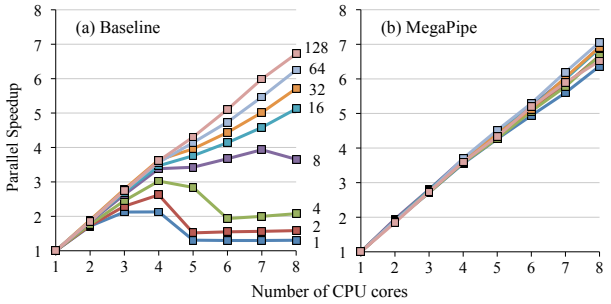

Figure 3: Comparison of parallel speedup for varying numbers of transactions per connection (labeled) over a range of CPU cores (x-axis) with 64 B messages.

	Number of transactions per connection							
	1	2	4	8	16	32	64	128
+P	211.6	207.5	181.3	83.5	38.9	29.5	17.2	8.8
P +B	18.8	22.8	72.4	44.6	31.8	30.4	27.3	19.8
PB +L	352.1	230.5	79.3	22.0	9.7	2.9	0.4	0.1
Total	582.4	460.8	333.1	150.1	80.4	62.8	45.0	28.7

Table 3: Accumulation of throughput improvement (%) over baseline, from three contributions of MegaPipe.

each) for a variety of transaction lengths. The baseline case on the left clearly shows that the scalability highly depends on the length of connections. For short connections, the throughput stagnates as core count grows due to the serialization at the shared accept queue, then suddenly collapses with more cores. We attribute the performance collapse to increased cache congestion and non-scalable locks [21]; note that the connection establishment process happens more frequently with short flows in our test, increasing the level of contention.

In contrast, the throughput of MegaPipe scales almost linearly regardless of connection length, showing speedup of 6.4 (for single-transaction connections) or higher. This improved scaling behavior of MegaPipe is mostly from the multi-core related optimizations techniques, namely partitioning and lwsocket. We observed similar speedup without batching, which enhances per-core throughput.

In Table 3, we present the incremental improvements (in percent over baseline) that Partitioning (P), Batching (B), and lwsocket (L) contribute to overall throughput, by accumulating each technique in that order. In this experiment, we used all eight cores, with 64 B messages (1 KiB messages yielded similar results). Both partitioning and lwsocket significantly improve the throughput of short connections, and their performance gain diminishes for longer connections since the both techniques act only at the connection establishment stage. For longer connections (not shown in the table), the gain from batching converged around 15%. Note that the case with partitioning alone (+P in the table) can be seen as *sockets with*

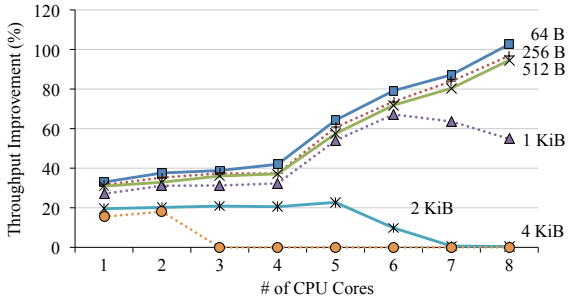

Figure 4: Relative performance improvement for varying message sizes over a range of CPU cores.

Affinity-Accept [33], as the both address the shared accept queue and connection affinity issues. lwsocket further contributes the performance of short connections, helping to achieve near-linear scalability as shown in Figure 3(b).

Lastly, we examine how the improvement changes by varying message sizes. Figure 4 depicts the relative throughput improvement, measured with 10-transaction connections. For the single-core case, where the improvement comes mostly from batching, MegaPipe outperforms the baseline case by 15–33%, showing higher effectiveness for small ($\leq$ 1 KiB) messages. The improvement goes higher as we have five or more cores, since the baseline case experiences more expensive off-chip cache and remote memory access, while MegaPipe effectively mitigates them with partitioning and lwsocket. The degradation of relative improvement from large messages with many cores reflects that the server was able to saturate the 10 G link. MegaPipe saturated the link with seven, five, and three cores for 1, 2, and 4 KiB messages, respectively. The baseline Linux saturated the link with seven and three cores for 2 and 4 KiB messages, respectively.

5.2 Macrobenchmark: memcached

We perform application-level macrobenchmarks of memcached, comparing the baseline performance to that of memcached adapted for MegaPipe as previously described. For baseline measurements, we used a patched[6] version of the stock memcached 1.4.13 release.

We used the `memaslap` [12] tool from libmemcached 1.0.6 to perform the benchmarks. We patched `memaslap` to accept a parameter designating the maximum number of requests to issue for a given TCP connection (upon which it closes the connection and reconnects to the server). Note that the typical usage of memcached is to use persistent connections to servers or UDP sockets, so the performance result from short connections may not be representative of memcached; rather, it should be inter-

[6] We discovered a performance bug in the stock memcached release as a consequence of unfairness towards servicing new connections, and we corrected this fairness bug.

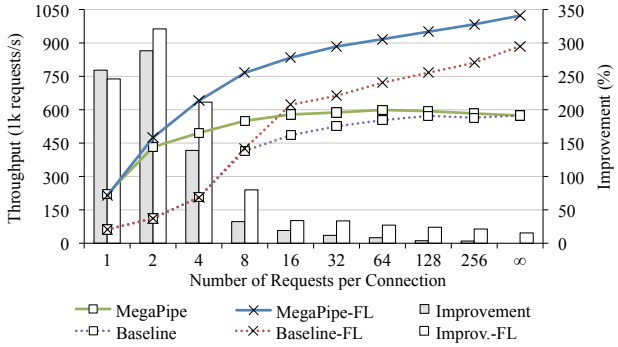

Figure 5: memcached throughput comparison with eight cores, by varying the number of requests per connection. ∞ indicates persistent connections. Lines with "X" markers (-FL) represent fine-grained-lock-only versions.

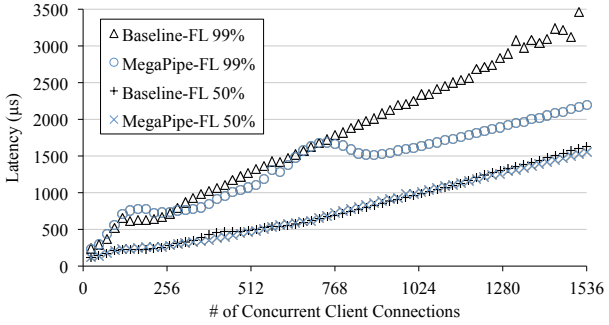

Figure 6: 50th and 99th percentile memcached latency.

preted as what-if scenarios for event-driven server applications with non-persistent connections.

The key-value workload used during our tests is the default `memaslap` workload: 64 B keys, 1 KiB values, and a get/set ratio of 9:1. For these benchmarks, each of three client machines established 256 concurrent connections to the server. On the server side, we set the memory size to 4 GiB. We also set the initial hash table size to 2^{22} (enough for 4 GiB memory with 1 KiB objects), so that the server would not exhibit performance fluctuations due to dynamic hash table expansion during the experiments.

Figure 5 compares the throughput between the baseline and MegaPipe versions of memcached (we discuss the "-FL" versions below), measured with all eight cores. We can see that MegaPipe greatly improves the throughput for short connections, mostly due to partitioning and lwsocket as we confirmed with the microbenchmark. However, the improvement quickly diminishes for longer connections, and for persistent connections, MegaPipe does not improve the throughput at all. Since the MegaPipe case shows about 16% higher throughput for the single-core case (not shown in the graph), it is clear that there is a performance-limiting bottleneck for the multi-core case. Profiling reveals that spin-lock contention takes roughly 50% of CPU cycles of the eight cores, highly limiting the scalability.

In memcached, normal get/set operations involve two locks: `item_locks` and a global lock `cache_lock`. The fine-grained `item_locks` (the number is dynamic, 8,192 locks on eight cores) keep the consistency of the object store from concurrent accesses by worker threads. On the other hand, the global `cache_lock` ensures that the hash table expansion process by the *maintenance thread* does not interfere with worker threads. While this global lock is inherently not scalable, it is unnecessary for our experiments since we configured the hash table expansion to not happen by giving a sufficiently large initial size.

We conducted experiments to see what would happen if we rule out the global lock, thus relying on the fine-grained locks (`item_locks`) only. We provide the results (with the suffix "-FL") also in Figure 5. Without the global lock, the both MegaPipe and baseline cases perform much better for long or persistent connections. For the persistent connection case, batching improved the throughput by 15% (note that only batching among techniques in §3 affects the performance of persistent connections). We can conclude two things from these experiments. First, MegaPipe improves the throughput of applications with short flows, and the improvement is fairly insensitive to the scalability of applications themselves. Second, MegaPipe might not be effective for poorly scalable applications, especially with long connections.

Lastly, we discuss how MegaPipe affects the latency of memcached. One potential concern with latency is that MegaPipe may add additional delay due to batching of I/O commands and notification events. To study the impact of MegaPipe on latency, we measured median and tail (99th percentile) latency observed by the clients, with varying numbers of persistent connections, and plotted these results in Figure 6. The results show that MegaPipe does not adversely affect the median latency. Interestingly, for the tail latency, MegaPipe slightly increases it with low concurrency (between 72–264) but greatly reduces it with high concurrency ($\geq$ 768). We do not fully understand these tail behaviors yet. One likely explanation for the latter is that batching becomes more effective with high concurrency; since that batching exploits parallelism from independent connections, high concurrency yields larger batch sizes.

In this paper, we conduct all experiments with the interrupt coalescing feature of the NIC. We briefly describe the impact of disabling it, to investigate if MegaPipe favorably or adversely interfere with interrupt coalescing. When disabled, the server yielded up to 50μs (median) and 200μs (tail) lower latency with low concurrency (thus underloaded). On the other hand, beyond near saturation point, disabling interrupt coalescing incurred significantly

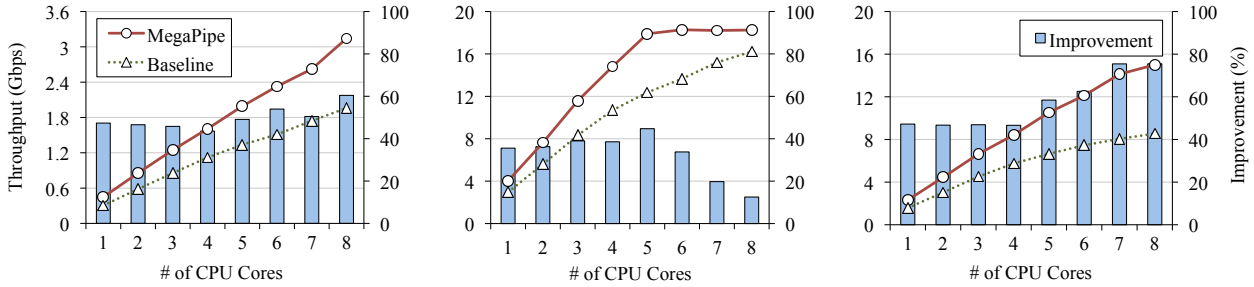

Figure 7: Evaluation of nginx throughput for the (a) SpecWeb, (b) Yahoo, and (c) Yahoo/2 workloads.

higher latency due to about 30% maximum throughput degradation, which causes high queueing delay. We observed these behaviors for both MegaPipe and baseline; we could not find any MegaPipe-specific behavior with interrupt coalescing in our experiments.

5.3 Macrobenchmark: nginx

Unlike memcached, the architecture of nginx is highly scalable on multi-core servers. Each worker process has an independent address space, and nothing is shared by the workers, so the performance-critical path is completely lockless. The only potential factor that limits scalability is the interface between the kernel and user, and we examine how MegaPipe improves the performance of nginx with such characteristics.

For the nginx HTTP benchmark, we conduct experiments with three workloads with static content, namely `SpecWeb`, `Yahoo`, and `Yahoo/2`. For all workloads, we configured nginx to serve files from memory rather than disks, to avoid disks being a bottleneck. We used weighttp[7] as a workload generator, and we modified it to support variable number of requests per connection.

SpecWeb: We test the same HTTP workload used in Affinity-Accept [33]. In this workload, each client connection initiates six HTTP requests. The content size ranges from 30 to 5,670 B (704 B on average), which is adopted from the static file set of SpecWeb 2009 Support Workload [9].

Yahoo: We used the HTTP trace collected from the Yahoo! CDN [13]. In this workload, the number of HTTP requests per connection ranges between 1 and 1,597. The distribution is heavily skewed towards short connections (98% of connections have ten or less requests, 2.3 on average), following the Zipf-like distribution. Content sizes range between 1 B and 253 MiB (12.5 KiB on average). HTTP responses larger than 60 KiB contribute roughly 50% of the total traffic.

Yahoo/2: Due to the large object size of the `Yahoo` workload, MegaPipe with only five cores saturates the two 10G links we used. For the `Yahoo/2` workload, we change the size of all files by half, to avoid the link bottleneck and observe the multi-core scalability behavior more clearly.

Web servers can be seen as one of the most promising applications of MegaPipe, since typical HTTP connections are short and carry small messages [13]. We present the measurement result in Figure 7 for each workload. For all three workloads, MegaPipe significantly improves the performance of both single-core and multi-core cases. MegaPipe with the `Yahoo/2` workload, for instance, improves the performance by 47% (single core) and 75% (eight cores), with a better parallel speedup (from 5.4 to 6.5) with eight cores. The small difference of improvement between the `Yahoo` and `Yahoo/2` cases, both of which have the same connection length, shows that MegaPipe is more beneficial with small message sizes.

6 Related Work

Scaling with Concurrency: Stateless event multiplexing APIs, such as `select()` or `poll()`, scale poorly as the number of concurrent connections grows since applications must declare the entire *interest set* of file descriptors to the kernel repeatedly. Banga et al. address this issue by introducing stateful interest sets with incremental updates [16], and we follow the same approach in this work with `mp_(un)register()`. The idea was realized with with `epoll` [8] in Linux (also used as the baseline in our evaluation) and `kqueue` [29] in FreeBSD. Note that this scalability issue in event delivery is orthogonal to the other scalability issue in the kernel: VFS overhead, which is addressed by lwsocket in MegaPipe.

Asynchronous I/O: Like MegaPipe, Lazy Asynchronous I/O (LAIO) [22] provides an interface with completion notifications, based on "continuation". LAIO achieves low overhead by exploiting the fact that most I/O operations do not block. MegaPipe adopts this idea, by processing non-blocking I/O operations immediately as explained in §4.1.

POSIX AIO defines functions for asynchronous I/O in UNIX [6]. POSIX AIO is not particularly designed for sockets, but rather, general files. For instance, it does

[7] `http://redmine.lighttpd.net/projects/weighttp/wiki`

not have an equivalent of `accept()` or `shutdown()`. Interestingly, it also supports a form of I/O batching: `lio_listio()` for AIO commands and `aio_suspend()` for their completion notifications. This batching must be explicitly arranged by programmers, while MegaPipe supports transparent batching.

Event Completion Framework [1] in Solaris and `kqueue` [29] in BSD expose similar interfaces (completion notification through a completion port) to MegaPipe (through a channel), when they are used in conjunction with POSIX AIO. These APIs associate individual AIO operations, not handles, with a channel to be notified. In contrast, a MegaPipe handle is a member of a particular channel for explicit partitioning between CPU cores. Windows IOCP [10] also has the concept of completion port and membership of handles. In IOCP, I/O commands are not batched, and handles are still shared by all CPU cores, rather than partitioned as lwsockets.

System Call Batching: While MegaPipe's batching was inspired by FlexSC [35, 36], the main focus of MegaPipe is I/O, not general system calls. FlexSC batches synchronous system call requests via asynchronous channels (syscall pages), while MegaPipe batches asynchronous I/O requests via synchronous channels (with traditional exception-based system calls). Loose coupling between system call invocation and its execution in FlexSC may lead poor cache locality on multi-core systems; for example, the `send()` system call invoked from one core may be executed on another, inducing expensive cache migration during the copy of the message buffer from user to kernel space. Compared with FlexSC, MegaPipe explicitly partitions cores to make sure that all processing of a flow is contained within a single core.

netmap [34] extensively use batching to amortize the cost of system calls, for high-performance, user-level packet I/O. MegaPipe follows the same approach, but its focus is generic I/O rather than raw sockets for low-level packet I/O.

Kernel-Level Network Applications: Some network applications are partly implemented in the kernel, tightly coupling performance-critical sections to the TCP/IP stack [25]. While this improves performance, it comes at a price of limited security, reliability, programmability, and portability. MegaPipe gives user applications lightweight mechanisms to interact with the TCP/IP stack for similar performance advantages, while retaining the benefits of user-level programming.

Multi-Core Scalability: Past research has shown that partitioning cores is critical for linear scalability of network I/O on multi-core systems [19, 20, 33, 38]. The main ideas are to maintain flow affinity and minimize unnecessary sharing between cores. In §3.4.1, we addressed the similarities and differences between Affinity-Accept [33] and MegaPipe. In [20], the authors address the scalability issues in VFS, namely inode and dentry, in the general context. We showed in §3.4.2 that the VFS overhead can be completely bypassed for network sockets in most cases.

The Chronos [26] work explores the case of direct coupling between NIC queues and application threads, in the context of multi-queue NIC and multi-core CPU environments. Unlike MegaPipe, Chronos bypasses the kernel, exposing NIC queues directly to user-space memory. While this does avoid in-kernel latency/scalability issues, it also loses the generality of TCP connection handling which is traditionally provided by the kernel.

Similarities in Abstraction: Common Communication Interface (CCI) [15] defines a portable interface to support various transports and network technologies, such as Infiniband and Cray's Gemini. While CCI and MegaPipe have different contexts in mind (user-level message-passing in HPC vs. general sockets via the kernel network stack), both have very similar interfaces. For example, CCI provides the *endpoint* abstraction as a channel between a virtual network instance and an application. Asynchronous I/O commands and notifications are passed through the channel with similar API semantics (e.g., `cci_get_event()`/`cci_send()` corresponding to `mp_dispatch()`/`mp_write()`).

The channel abstraction of MegaPipe shares some similarities with Mach port [11] and other IPC mechanisms in microkernel designs, as it forms queues for typed messages (I/O commands and notifications in MegaPipe) between subsystems. Especially, Barrelfish [17] leverages message passing (rather than sharing) based on event-driven programming model to solve scalability issues, while its focus is mostly on inter-core communication rather than strict intra-core communication in MegaPipe.

7 Conclusion

Message-oriented network workloads, where connections are short and/or message sizes are small, are CPU-intensive and scale poorly on multi-core systems with the BSD Socket API. In this paper, we introduced MegaPipe, a new programming interface for high-performance networking I/O. MegaPipe exploits many performance optimization opportunities that were previously hindered by existing network API semantics, while being still simple and applicable to existing event-driven servers with moderate efforts. Evaluation through microbenchmarks, memcached, and nginx showed significant improvements, in terms of both single-core performance and parallel speedup on an eight-core system.

Acknowledgements

We thank Luca Niccolini, members of NetSys Lab at UC Berkeley, anonymous OSDI reviewers, and our shepherd Jeffrey Mogul for their help and invaluable feedback. The early stage of this work was done in collaboration with Keon Jang, Sue Moon, and KyoungSoo Park, when the first author was affiliated with KAIST.

References

[1] Event Completion Framework for Solaris. `http://developers.sun.com/solaris/articles/event_completion.html`.

[2] Intel 10 Gigabit Ethernet Adapter. `http://e1000.sourceforge.net/`.

[3] memcached - a distributed memory object caching system. `http://memcached.org/`.

[4] Node.js: an event-driven I/O server-side JavaScript environment. `http://nodejs.org`.

[5] Receive-Side Scaling. `http://www.microsoft.com/whdc/device/network/ndis_rss.mspx`, 2008.

[6] The Open Group Base Specifications Issue 7. `http://pubs.opengroup.org/onlinepubs/9699919799/`, 2008.

[7] Intel 8259x 10G Ethernet Controller. Intel 82599 10 GbE Controller Datasheet, 2009.

[8] epoll - I/O event notification facility. `http://www.kernel.org/doc/man-pages/online/pages/man4/epoll.4.html`, 2010.

[9] SPECweb2009 Release 1.20 Support Workload Design Document. `http://www.spec.org/web2009/docs/design/SupportDesign.html`, 2010.

[10] Windows I/O Completion Ports. `http://msdn.microsoft.com/en-us/library/windows/desktop/aa365198(v=vs.85).aspx`, 2012.

[11] ACCETTA, M., BARON, R., BOLOSKY, W., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. Mach: A New Kernel Foundation For UNIX Development. In *Proc. of USENIX Summer* (1986).

[12] AKER, B. memaslap: Load testing and benchmarking a server. `http://docs.libmemcached.org/memaslap.html`, 2012.

[13] AL-FARES, M., ELMELEEGY, K., REED, B., AND GASHINSKY, I. Overclocking the Yahoo! CDN for Faster Web Page Loads. In *Proc. of ACM IMC* (2011).

[14] ARMSTRONG, J., VIRDING, R., WIKSTRÖM, C., AND WILLIAMS, M. *Concurrent Programming in Erlang*, second ed. Prentice Hall, 1996.

[15] ATCHLEY, S., DILLOW, D., SHIPMAN, G., GEOFFRAY, P., SQUYRES, J. M., BOSILCA, G., AND MINNICH, R. The Common Communication Interface (CCI). In *Proc. of IEEE HOTI* (2011).

[16] BANGA, G., MOGUL, J. C., AND DRUSCHEL, P. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *Proc. of USENIX ATC* (1999).

[17] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: A new OS architecture for scalable multicore systems. In *Proc. of ACM SOSP* (2009).

[18] BILENKO, D. gevent: A coroutine-based network library for Python. `http://www.gevent.org/`.

[19] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An Operating System for Many Cores. In *Proc. of USENIX OSDI* (2008).

[20] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An Analysis of Linux Scalability to Many Cores. In *Proc. of USENIX OSDI* (2010).

[21] BOYD-WICKIZER, S., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. Non-scalable locks are dangerous. In *Proc. of the Linux Symposium* (July 2012).

[22] ELMELEEGY, K., CHANDA, A., COX, A. L., AND ZWAENEPOEL, W. Lazy Asynchronous I/O For Event-Driven Servers. In *Proc. of USENIX ATC* (2004).

[23] ENGELSCHALL, R. S. Portable Multithreading - The Signal Stack Trick for User-Space Thread Creation. In *Proc. of USENIX ATC* (2000).

[24] HERBERT, T. RPS: Receive Packet Steering. `http://lwn.net/Articles/361440/`, 2009.

[25] JOUBERT, P., KING, R. B., NEVES, R., RUSSINOVICH, M., AND TRACEY, J. M. High-Performance Memory-Based Web Servers: Kernel and User-Space Performance. In *Proc. of USENIX ATC* (2001).

[26] KAPOOR, R., PORTER, G., TEWARI, M., VOELKER, G. M., AND AMIN, V. Reducing Datacenter Application Latency with Endhost NIC Support. Tech. Rep. CS2012-0977, UCSD, April 2012.

[27] KLEIMAN, S. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proc. of USENIX Summer* (1986).

[28] KROHN, M., KOHLER, E., AND KAASHOEK, M. F. Events Can Make Sense. In *Proc. of USENIX ATC* (2007).

[29] LEMON, J. Kqueue: A generic and scalable event notification facility. In *Proc. of USENIX ATC* (2001).

[30] MATHEWSON, N., AND PROVOS, N. libevent - an event notification library. `http://libevent.org`.

[31] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND RYAN, S. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review 43*, 4 (2010), 92–105.

[32] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. Flash: An Efficient and Portable Web Server. In *Proc. of USENIX ATC* (1999).

[33] PESTEREV, A., STRAUSS, J., ZELDOVICH, N., AND MORRIS, R. T. Improving Network Connection Locality on Multicore Systems. In *Proc. of ACM EuroSys* (2012).

[34] RIZZO, L. netmap: a novel framework for fast packet I/O. In *Proc. of USENIX ATC* (2012).

[35] SOARES, L., AND STUMM, M. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proc. of USENIX OSDI* (2010).

[36] SOARES, L., AND STUMM, M. Exception-Less System Calls for Event-Driven Servers. In *Proc. of USENIX ATC* (2011).

[37] SYSOEV, I. nginx web server. `http://nginx.org/`.

[38] VEAL, B., AND FOONG, A. Performance Scalability of a Multi-Core Web Server. In *Proc. of ACM/IEEE ANCS* (2007).

[39] VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G. C., AND BREWER, E. Capriccio: Scalable Threads for Internet Services. In *Proc. of ACM SOSP* (2003).

[40] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proc. of ACM SOSP* (2001).