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

Paper #40, 12 Pages

Abstract

We present MegaPipe, a new API for efficient, scalable network I/O. The design of MegaPipe centers around the abstraction of a *channel* – a per-core, bidirectional pipe between the kernel and user space, used to efficiently exchange both I/O requests and event notifications. We implement MegaPipe in Linux and adapt the popular memcached and nginx applications to use MegaPipe. 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 microbenchmark tests on an 8-core server with 64 byte messages, MegaPipe outperforms baseline Linux by between 23% (for long connections) and 604% (for short connections). For similar parameters, MegaPipe improves the performance of memcached by between 1.5% and 268%. For a workload based on a real-world traffic traces, MegaPipe improves the performance of nginx by 67.6%.

1 Introduction

Existing network stacks on multi-core systems have difficulty scaling to high connection rates and are inefficient for "message-oriented" workloads, by which we mean workloads with short connections and/or small messages.[1] Such message-oriented workloads include HTTP, RPC, key-value stores with small objects (e.g., RAMCloud [27]), 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, either (*i*) modifying the kernel stack but leaving the APIs untouched [29–31], or (*ii*) adding new APIs to complement the existing APIs [5]. While this approach has 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, and so forth – limits the extent to which it can be optimized for performance. In contrast, a clean-slate redesign would offer the opportunity to present an API that is specialized for high performance network I/O.

[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.

However 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 would offer the opportunity to design 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:

Lightweight sockets: sockets today are represented by file descriptors and hence inherit certain unnecessary file-related overheads. MegaPipe instead introduces lwsocket, a lightweight socket abstraction that is core-local and not wrapped in file-related data structures.

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

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

MegaPipe thus offers an API that is both streamlined (i.e., selecting and implementing only a specialized subset of the general features of the BSD Socket API) and simplified (i.e., combining network connections, asynchronous I/O, and completion notifications under the roof of a single unified abstraction).

We implement MegaPipe in Linux and adapt two popular applications – memcached [8] and the nginx web server [10] – to use MegaPipe. In our microbenchmark tests on an 8-core server with 64 byte messages, we show that MegaPipe outperforms the baseline Linux networking stack by between 23% (for long connections) and 604% (for short connections). For similar parameters, MegaPipe improves the performance of memcached by between 1.5% and 268%. For a workload based on a real-world traffic traces, MegaPipe improves the performance of nginx by 67.6%.

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, review related work in §6 and finally conclude.

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 identify limitations of the current Linux networking stack (§2.1) and then quantify these assertions with a series of RPC-like microbenchmark experiments (§2.2).

To provide context for our upcoming discussion, we first briefly describe the life of a Linux TCP network connection. A connection begins when a client issues a SYN packet to a server with a listening socket. As the TCP 3-way handshake process completes, the Linux kernel initializes a TCP Control Block (TCB) to track the TCP state of the new connection and enqueues it on a single queue provided by the server's listening socket. Once an application thread invokes the `accept()` system call, the new connection is dequeued, and is issued a file descriptor (FD) for subsequent use by the application thread. The application thread completes I/O operations using the FD along with system calls such as `read()` and `write`. Finally, the application thread closes the connection with the `close()` system call, and the kernel releases resources associated with the corresponding FD.

2.1 Performance Limitations

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

System Calls: Previous work has shown that system calls from user-space into the kernel are expensive [30] and can significantly impact performance. This performance overhead is exacerbated for message-oriented workloads with small messages that result in a large number of `read()` and `write()` operations.

File Descriptors: In POSIX operating systems, network sockets are abstracted as regular file descriptors (FDs), just like other file types: disk files, pipes, and devices. While this layer of abstraction provides programmers a unified interface, it also adds various unnecessary layers of overhead (we elaborate on these overheads in §3.4.1). The overhead due to FDs is general, occuring even in a single core context. The problem is exacerbated however by the fact that these entries are globally visible in the filesystem and hence synchronization is required each time a file descriptor (and thus a socket) is created. Further, as FD numbers must be unique per-process (rather than per-thread), the search for the next available FD number requires a process-wide lock. For message-oriented workloads with short connections, where sockets are frequently opened as new connections arrive, servers quickly become overloaded since process-wide locking due to FD allocation and system-wide synchronization due to globally-visible VFS structures prevents concurrency.

Single `accept()` Queue: As explained in previous work [22,29], a single listening socket with single connection `accept()` backlog queue (and associated exclusive lock) forces CPU cores to serialize queue access requests; this negatively impacts the performance of both producers (kernel threads) enqueuing new connections and consumers (application threads) accepting new connections. The inefficiencies due to a single accept queue thus limit performance scaling with multiple cores.

Cache Locality: As a best practice, modern operating systems provide two sets of technologies for balancing network I/O across multiple cores. In the case of unbound connections (incomplete TCP 3-way handshake, or not yet claimed by an application thread), incoming packets are distributed across CPU cores on a flow basis, either by hardware [11] or software [13]. In the case of bound connections, again either hardware [17] or software [12] maintains a table that maps flows to cores, thereby ensuring that (in-kernel) packet processing occurs on the same core as the application thread handling the connection. However, CPU core distribution of both unbound and bound connections is random. Thus, throughout a single connection's lifespan, processing likely occurs on two different cores, and this causes poor cache locality of the TCB; a TCB entry is initialized on one core when the connection is established, but migrates to a second core once an application thread issues system calls for further RX/TX processing.

2.2 Performance Limitations Quantified

We quantify the performance impact due to the above limitations with a series of microbenchmarks. As we noted before, the above inefficiencies manifest themselves primarily in workloads that involve *short connections* or *small-sized messages*, particularly with increasing server scalability. Our microbenchmark tests thus focus on these problematic scenarios and we craft our experiments to highlight the impact for each in isolation.

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. We refer to a single request-response exchange as a *transaction*. Default parameters are 64 bytes per message and 10 transactions per connection. Messages are random octets, and neither the client nor server interprets their contents. Each client maintains 512 concurrent connections which we determined ensures that traffic generation is never the performance limiter. 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 [18] 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 two 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 [3]. All systems have an Intel 82599 10G Ethernet adapter, 12 GB of DRAM, and two Intel Xeon X5560 processors, each of which has four 2.80 GHz cores.

Performance with Short Connections: As described earlier, 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 excessive data transfer and hence this workload serves to highlight the overhead due to costly system calls and FD initialization.

We measure how connection lifespan affects overall system performance by varying the number of transactions per connection. The left plot in Figure 1 shows throughput measured in transactions per second. Total throughput is significantly lower with relatively few (1-8) transactions per connection. The cost of connection establishment eventually becomes negligible for 128+ transactions per connection, and we observe that throughput in single-transaction connections is roughly 22 times lower than that of long connections!

In conclusion, we see that the overhead due to system calls and FD initialization can significantly impact workloads with high rates of short connections.

Performance with Small Messages: Small messages result in greater relative network I/O overhead in comparison to larger messages. In fact, overhead remains roughly constant and thus, independent of message size; in comparison with a 1-byte message, a 10 KB message adds minimal additional overhead due to the copy between user-space and the kernel, and formation of the packet itself. Thus, techniques such as TCP Segmentation Offload, Jumbo Frames, and Large Receive Offload, which effectively reduce CPU burden by batch-processing large payloads, do not help these workloads. Rather, it is the fixed cost of system calls, TCP/IP processing, and device driver operations that dominate network I/O overhead for these workloads.

To measure these effects, we perform a second microbenchmark with response sizes varying from 64 bytes to 64 KB (varying the request size as well had negligible effects). The second plot in Figure 1 shows the measured throughput (in Gbps) and CPU usage (measured with `mpstat` [19]) for various message sizes. For small messages ($\leq$ 2 KB) network I/O does not even saturate the 10G link. For medium-sized messages (2 KB and 4 KB), CPU utilization is extremely high, leaving few CPU cycles for further application processing. In conclusion, we see that the fixed overhead due to system calls significantly impacts workloads with small messages.

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 the plot on the right of Figure 1, we plot 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 bars labeled "Ideal" represent 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 is avoided up to this point. Beyond four cores, the marginal performance gain with each additional core

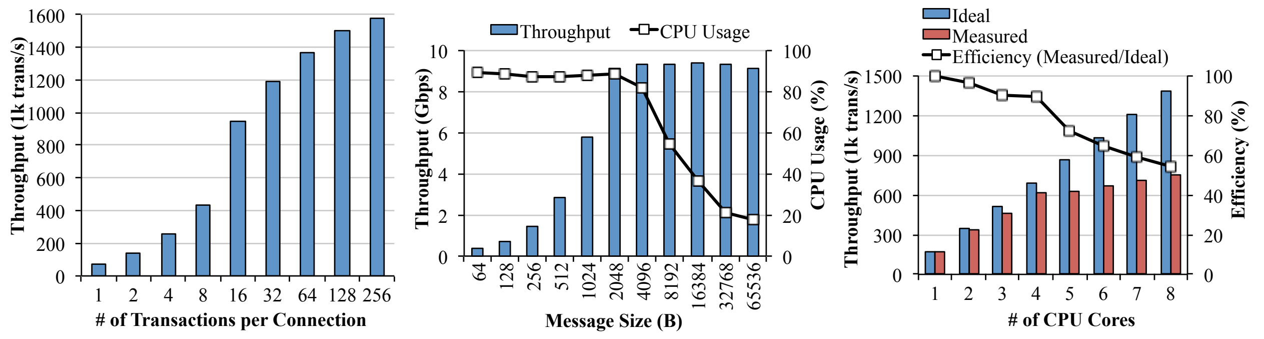


Figure 1: The negative impact of connection lifespan, the negative impact of message size, and poor multi-core scalability, respectively.

quickly diminishes, and with eight cores, speedup is only 4.3. 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 depends on connection duration, further confirming the results in Fig. 1 (to the left). With only one transaction per connection (instead of the default 10 used in the rightmost plot of Fig. 1), speedup with eight cores was 1.3, while with longer connections of 128 transactions, performance scaled better with a speedup of 6.4.

In conclusion, these results show that serialization due to the single `accept()` queue, TCB cache migration due to a single connection being processed by two CPU cores, and the costly process of locating and allocating a FD all negatively impact our ability to scale connection processing with multiple cores.

3 MegaPipe Design

MegaPipe is a new programming interface for high-performance network I/O that addresses the inefficiencies highlighted in the previous section while providing 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 The rationale for a new API

The state of the art in event-driven server programming has evolved incrementally over the years. In Unix-like operating systems, a socket is represented as a regular file. The blocking semantics of the Berkeley Socket API forced a thread (or even a process) to handle only a single connection with one socket. However, to deal with high concurrency without thread overheads, event-driven servers require non-blocking socket semantics to multiplex concurrent connections on a single thread. Supporting this led to the introduction of additional mechanisms – e.g., `epoll`, `kqueue`, or event completion ports – used to check the readiness of a socket. This approach based on additional abstractions has the clear advantage of code reuse and backward compatibility. However, the drawback of retaining legacy APIs and abstractions is that it limits many opportunities for optimization. For instance: *i)* sockets represented as files inherit the overheads of files in the kernel; *ii)* it is difficult to aggregate multiple I/O requests from concurrent connections to amortize system call overheads. In this paper, we instead ask what a new API designed with the primary goal of supporting network I/O would look like?

MegaPipe is a new network programming API that exposes more opportunities for optimization and more control to applications, while being easy to use.

3.2 Design Goals

MegaPipe has two main goals: high performance and ease of use.

High Performance: We design MegaPipe with multi-core systems in mind. Performance on multi-core systems depends on two factors: single core performance and performance scaling with multiple cores. For the single-core case, we try to minimize the high per-connection and per-message costs. For the multi-core case, we offer applications an explicit way to (*i*) partition a workload across individual cores, (*ii*) "affinitize" a connection to a core, and (*iii*) minimize the need for synchronization between cores, to achieve near-linear scaling with core count.

Simplicity and Applicability: We design MegaPipe to be conceptually simple and self-contained (not dependent on existing APIs). MegaPipe should be applicable to a variety of event-driven mechanisms, such as callback, event loop, or lightweight threading model. Since we introduce a new API, existing server applications need modifications to benefit from MegaPipe. In Section 4, we describe our experience modifying memcached and nginx, demonstrating that existing event-driven servers can be adapted to use MegaPipe with minimal modifications.

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 an application interacts with the kernel via *MegaPipe channels*. With MegaPipe, an application thread running on each core[2] opens a separate channel for communication between the kernel and user-space.

Each channel multiplexes its own set of *handles* (sockets and other file types) for their asynchronous I/O requests and completion notification events. The application thread associates a handle with the corresponding channel, for either a regular file descriptor or a lightweight socket, *lwsocket* (Section 3.4.1). Once the application thread registers a handle, any further operations on the handle are done through the channel, for the lifespan of the connection or until the handle is explicitly unregistered. When a listening socket is registered, MegaPipe duplicates a new listen backlog queue, which runs independently from the original listening socket and is responsible for new connections for the core (Section 3.4.2). Since each channel has a disjoint set of handles, all I/O processing is *partitioned* across individual channels, without requiring any synchronization between cores.

Each channel is composed of two message streams: a request stream and a completion stream. A sample series of messages is shown on the right side of the figure. 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* manner, to minimize the context switch between user and kernel (Section 3.4.3).

3.4 Design Components

3.4.1 lwsocket: Lightweight Socket

`accept()`ing an established connection is an expensive process in the context of the existing Virtual File System (VFS). As mentioned, in Unix-like operating systems, many different types of open files (disk files, sockets, pipes, devices, etc.) are identified as a *file descriptor*. A file descriptor is an integer identifier used as an indirect reference to an opened *file instance*, which maintains the status of the opened file. Multiple file instances may point to the same *inode*, which represents a unique, permanent file object. For a socket-type inode, it points to an actual socket, such as a TCP connection object.

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. For network sockets, however, we claim that these layers of abstraction can be overkill for the following reasons:

(1) FD allocation. The POSIX standard requires that a newly allocated file descriptor be the lowest integer not currently used by the process. 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, limiting the scalability of multi-threaded applications.

(2) Shared access. 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 (CGI process redirected to a HTTP socket is an exception) and not opened multiple times, this indirection is typically unnecessary.

(3) FD lifetime. Unlike permanent disk-backed files, network sockets are ephemeral. Every time a new connection is established or torn down, the inode object is newly allocated and freed rather than being cached. This is exacerbated on multi-core systems since the kernel maintains the inode and dentry as globally visible data structures [22].

(4) Function call overhead. The many layers of abstraction require an excessive number of function calls to perform the actual I/O operation. For instance, a `read()` system call on a TCP socket requires eight nested function calls to reach `tcp_recvmsg()`. While function calls are cheap on modern CPUs, the overhead becomes significant considering that busy network servers may perform millions of network I/O operations, unlike disk I/O.

To address the above issues, we propose lightweight sockets – *lwsocket*. Unlike regular files, a lwsocket is identified by an arbitrary integer, not the lowest possible integer. lwsocket is a common-case optimization for network connections; it does not create a corresponding file instance, inode, or dentry. A lwsocket is only locally visible within the associated MegaPipe channel, to avoid

[2]For the sake of concise explanation, we assume one-to-one match between cores and threads. In fact, MegaPipe can be used in various configurations, such as one thread for each CPU die or one thread for each NUMA node.

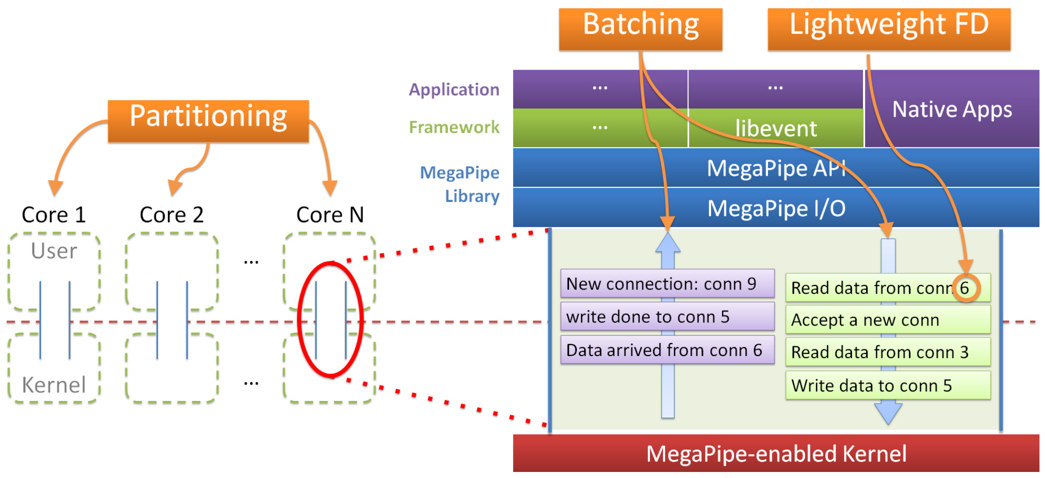


Figure 2: MegaPipe architecture

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 such cases, and in cases where the full generality of file descriptors is required, MegaPipe provides a fallback option to convert a lwsocket into a regular file.

3.4.2 Listening Socket Partitioning

In multi-threaded server applications, threads share the same listening socket. As described in Section 2.1, a listening socket shared by multiple threads causes three issues: (*i*) serialization due to an explicit lock, (*ii*) cache line contention on the listening socket between cores, and (*iii*) TCB migration from one CPU cache (where a connection is established) to another (where the connection is actually used onwards).

In the recent work on Affinity-Accept, a listening socket has per-core backlog queues and application threads that call `accept()` prioritize their local backlog queue [29]. With Affinity-Accept, connection processing becomes completely parallelizable and independent (unless the local queue is empty), and connections established on one core are processed on the same core. Affinity-Accept requires no modification to user applications (assuming application threads already parallelize connection acceptance).

In MegaPipe, we achieve essentially the same goals but with a somewhat different approach. When an application thread associates a listening socket to a channel, MegaPipe clones a separate listening socket. The new listening socket has its own backlog queue which is only responsible for connections established on a particular subset of cores, that are explicitly specified by an optional `cpumask` parameter. When a shared listening socket is registered to MegaPipe channels with disjoint `cpumask` parameters, all channels (and thus cores) have completely partitioned backlog queues.

We briefly discuss two main differences between our technique and Affinity-Accept. First, our technique requires user-level applications to partition a listening socket explicitly, rather than transparently done. The downside is that legacy applications do not benefit. However, explicit partitioning provides more flexibility for user applications (e.g., to forego partitioning for single-thread applications, to establish one backlog queue for each physical core in SMT systems, *etc.*)

Second, MegaPipe does not directly address how load is balanced across backlog queues. The issue is well studied in Affinity-Accept, by allowing idle cores to remotely accept a new connection from a busy core [29]. MegaPipe expects server applications to apply a similar technique, but at the user level. While this adds a burden to programmers, we argue that applications have context regarding load – e.g., number of concurrent connections (processed *and* pending) and connection semantics – that is invisible to the kernel and that goes beyond the number of pending connections in backlog queues as used Affinity-Accept. To support applications in balancing load, the MegaPipe interface supplies applications with the current number of pending connections in backlog queues and supports a mechanism to migrate a handle from one channel to another.

Function	Description
`mp_create()`	Creates a new MegaPipe channel instance.
`mp_register()`	Creates 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_read()` `mp_write()` `mp_disconnect()`	Issues an asynchronous request for the provided I/O operation. Once MegaPipe has completed the request, the completion notification will be made available via the `mp_dispatch()` function. Each I/O command has its own required command-specific arguments.
`mp_accept()`	Accepts a new connection from a given listening handle asynchronously. Applications specifies whether to accept a connection as a regular socket or a lwsocket.
`mp_dispatch()`	Retrieves a single completion notification for the given channel from the kernel.

Table 1: MegaPipe API functions (not exhaustive).

3.4.3 System Call Batching

Recent research efforts report that system calls are expensive not only due to the cost of switching modes, but also because of the negative effect on IPC for both user and kernel [30]. 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 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*) a timer expires, (*iii*) there are not any more requests to be issued at that time, or (*iv*) the application explicitly calls `mp_flush()`, then the collected requests are passed 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 notifications from kernel in a batch.

We found that a batch size beyond 8-16 offers little marginal performance gain, and we set the threshold in our implementation to 32 by default (adjustable). One potential concern is that batching may affect the latency of network servers since I/O requests are queued. However, batching happens only when the server is overloaded, so it does not affect latency when underloaded. Even if the server is overloaded, the additional latency should be minimal because the batch threshold is fairly small, compared to the large queues (thousands of packets) of modern NICs [17].

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

while true:
  ev = mp_dispatch(ch)
  conn = conn_set.find(ev.cookie)
  if ev.cmd == ACCEPT:
    mp_accept(conn.handle)
    conn = conn_set.new()
    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)
    conn_set.delete(conn)
```

Listing 1: Pseudocode for ping-pong server event loop

3.5 API

The MegaPipe user-level library provides a set of API functions to server applications, to hide the complexity of batching and the actual implementation of MegaPipe. Table 1 lists key API functions. Applications can issue I/O operations with either a regular file descriptor or a lwsocket.

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 original version of nginx calls 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.

We present pseudocode of a simple pingpong server to illustrate how applications use MegaPipe, in Listing 1.

An application thread initially creates a MegaPipe channel and registers a listening socket with `cpumask` 0x01 (first bit is set) which means 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.

Like other event-driven servers, our server has an event loop which pumps out new (completion notification) events from the channel. Recall that in MegaPipe an event represents a completion notification rather than readiness (as in `epoll`, for instance). The inside of the event loop is fairly simple: given an event, the application fires a new asynchronous I/O operation. Compared to a model based on readiness notification, MegaPipe's completion notifications greatly simplifies the complexity of I/O multiplexing; applications can blindly issue I/O operations without tracking the readiness of concurrent connections.

Currently MegaPipe provides only I/O-related functionality. In future work, we plan to support general system calls in an asynchronous fashion. For example, we can invoke the `futex()` system call, which is not related to file operations, to asynchronously grab a lock. We are currently working on implementing such functionality with `mp_syscall(syscall number, arg1, ...)`, to support general system calls in MegaPipe.

3.6 Discussion: Thread-Based Architecture

The current MegaPipe design naturally fits event-driven servers with an event loop and callback or state machine mechanisms [28, 33]. In contrast, it is not straightforward for MegaPipe to support event-driven servers in which each thread serves one connection. This is because threaded servers tend to directly utilize socket APIs rather than network I/O frameworks. Further, the blocking behavior of threads scheduled by kernel does not give user applications precise control of concurrency, blocking, and scheduling.

Interestingly, some thread frameworks based on user-level threading [2, 21] can benefit from MegaPipe. These frameworks manage massively-concurrent user-level threads on top of one kernel thread to avoid the overhead of kernel threads while retaining the advantages of the threaded servers. The frameworks intercept I/O calls issued by user threads to keep the kernel thread (the entire set of user threads) from blocking, and manage the outstanding I/O requests with asynchronous I/O mechanisms, such as `epoll` and `kqueue`. As this underlying I/O is done in an event-driven manner, MegaPipe can be easily integrated without any modifications of applications on top of those frameworks.

4 Implementation

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

4.1 MegaPipe Kernel/User Implementation

Kernel Implementation: The MegaPipe kernel implementation interacts with user applications as a special device, `/dev/megapipe`. The MegaPipe user-level library opens this file to create a channel, and invokes `ioctl()` system calls to issue I/O requests and dispatch completion notifications for that channel.

The kernel maintains a set of handles for both regular FDs and lwsockets in a red-black tree for each channel. Unlike a per-process file table, each channel is only accessed by one thread, avoiding data sharing between threads (thus cores). MegaPipe 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 (this range is used since it is unlikely these values will conflict with regular FD numbers, since the POSIX standard allocates the lowest possible integer for FDs [15]).

MegaPipe currently supports the following file types: sockets, pipes, and FIFOs. MegaPipe handles asynchronous I/O requests differently based on the file type. For sockets (such as TCP, UDP, and UNIX domain), MegaPipe utilizes the native callback interface supported by kernel sockets for optimal performance. For other file types, MegaPipe internally emulates asynchronous I/O with `epoll` and non-blocking VFS operations within kernel.

We implemented MegaPipe 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) to the Linux kernel itself, due to the following issues: (*i*) we modified `epoll` exposes its API to not only user but also kernel module; (*ii*) we modified the Linux kernel to allow multiple sockets to listen on the same port, which traditionally is not allowed; and (*iii*) we also enhanced the socket lookup process for incoming packets to consider `cpumask` when choosing a destination socket.

User-Level Library:: The MegaPipe user-level library is essentially a simple wrapper of the kernel module, and it is written in about 400 lines of code. The MegaPipe user library performs two main roles: (*i*) transparently provid-

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

Table 2: Lines of code for application adaptations

ing batching for asynchronous I/O requests and their completion notifications. (*ii*) performing communication with the kernel module via the `ioctl()` system call.

Current Limitations:: MegaPipe does not yet support disk files. The current Linux VFS only supports asynchronous I/O for unbuffered files (opened with `O_DIRECT`, which skips the buffer cache) [4]. When a file block for a buffered file is missing in the buffer cache, the I/O operation blocks. To work around this issue, we plan to adopt a lightweight technique made in [30]; when an I/O operation is about to block, we spawn a new thread on demand to handle the blocking request, while the current thread continues.

4.2 Application Integration

We adapted two popular event-driven servers, memcached 1.4.13 [8] (an in-memory key-value store) and nginx 1.0.15 [10] (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 challenges during the adaptation process. We briefly introduce our experiences here, and show the performance benefits in Section 5.

memcached: memcached uses the libevent [6] 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 simple tasks[3]:

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.

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

[3]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.

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 were split into two states: actions prior to the MegaPipe function call, and actions that follow the MegaPipe function call and depend on its result.

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 [6]; (*ii*) nginx was designed to support not only the readiness model, but also the completion notification model, which nicely fits with MegaPipe. (*iii*) all worker processes already accept new connections in parallel.

nginx has an extensible *event module* architecture, which allows drop-in replacement for its underlying event-driven model. Under this architecture, we implemented a MegaPipe event module and registered `mp_read()` and `mp_writev()` as the actual I/O functions. While all worker processes accepted connections, the MegaPipe adaptation adds partitioning to the shared listening socket.

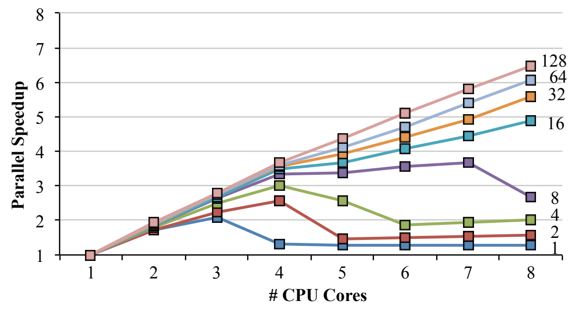
5 Evaluation

We evaluated the performance gains yielded by MegaPipe both through a collection of microbenchmarks, akin to those presented at the end of Section 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, with one server, two clients, and one 10G interface per system) as described in Section 2.2.

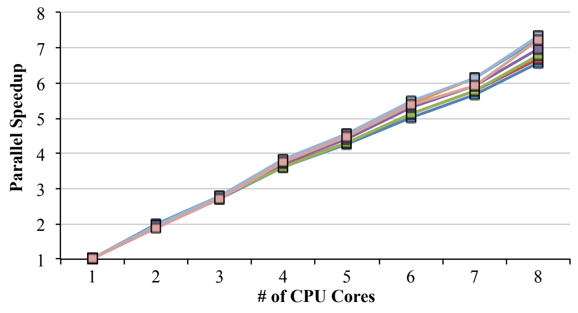
5.1 Microbenchmarks

The purpose of the microbenchmark results is three-fold. First, utilization of the same benchmark strategy as in Section 2 allows for direct evaluation of the low-level limitations we previously identified. Second, these microbenchmarks give us the opportunity to measure an upper-bound on performance, as macrobenchmarks may not always illustrate a performance upper-bound due to application overhead. Third, microbenchmarks allow us to illuminate the performance contributions of each of MegaPipe's individual components.

We begin with a pair of microbenchmarks similar in nature to those displayed in Figure 1. Figure 3 provides a side-by-side comparison of speedup for a variety of transaction lengths, while Figure 4 shows a side-by-side com-

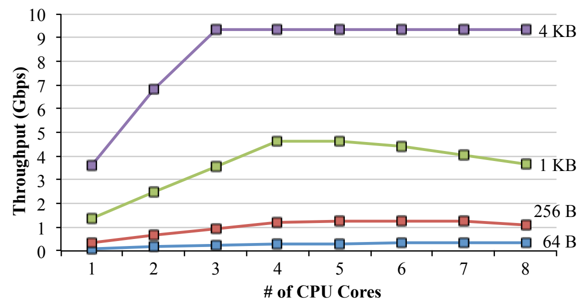


(a) Baseline

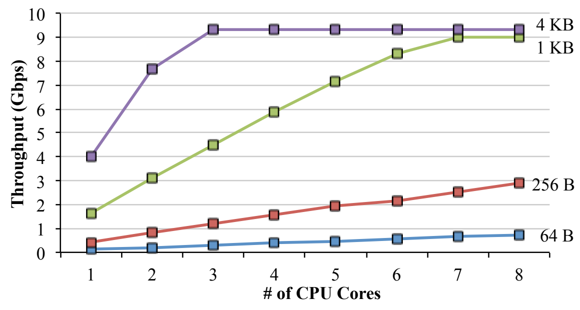


(b) MegaPipe

Figure 3: Comparison of speedup for varying numbers of transactions per connection over a range of CPU cores with 64 Byte messages.

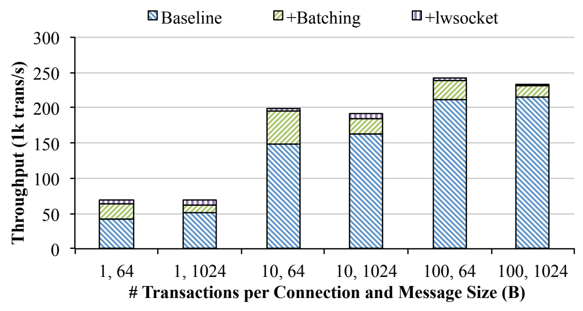


(a) Baseline

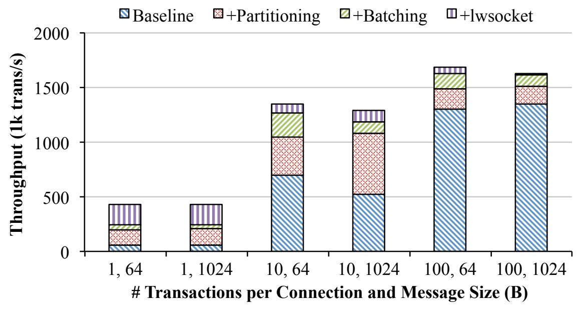


(b) MegaPipe

Figure 4: Comparison of throughput for varying message sizes over a range of CPU cores with 10 transactions per connection.



(a) 1 CPU Core



(b) 8 CPU Cores

Figure 5: Breakdown of individual MegaPipe performance improvement contributions.

parison of measured throughput for a variety of message sizes. Comparing Figure 3(b) to Figure 3(a), it is clear that MegaPipe scales well for connections of any length, unlike the baseline (i.e., unmodified Linux). Similarly comparing Figure 4(b) with Figure 4(a) reveals that MegaPipe achieves greater throughput for small message sizes, and is even able to saturate the 10G link with 1 KB messages while the baseline, unmodified Linux, cannot.

The final microbenchmark we present provides a breakdown of the performance benefits from each of MegaPipe's contributions. In Figure 5, we present a breakdown of the improvements that *Partitioning*, *Batch-*

ing, and *lwsocket* each contribute to overall throughput. For the single-core case the most performance gain comes from batching. For the multi-core case, we can see that the performance improvement with lwsocket highly depends on the lifespan of connections.

5.2 Macrobenchmark: memcached

We perform application-level macrobenchmarks of memcached, comparing baseline performance to that of memcached adapted for MegaPipe as previously described. For baseline measurements, we used a patched[4] version of the stock memcached 1.4.13 release.

We used the `memaslap` [7] 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 connection (upon which it closes the connection socket and reconnects to the server).

The key-value workload used during our tests is the default `memaslap` workload: 64 byte keys, 1 KB values, and a get/set ratio of 9:1. For these benchmarks, each of two client machines established 512 concurrent connections to the server.

Figure 6(a) compares the throughput between baseline and MegaPipe versions of memcached for the case of 4 requests per connection. It is clear that as the second CPU socket becomes involved ($>$ 4 cores), baseline performance suffers due to both increased contention for the single `accept()` queue and synchronization delays due to VFS structures and FD initialization. In fact, for 8 cores, MegaPipe memcached throughput is over 125% improved compared to the baseline.

Figure 6(b) showcases the great throughput improvements that MegaPipe yields for short connections. In fact, for connections with a single request, throughput of MegaPipe memcached throughput is over 250% improved compared to the baseline.

5.3 Macrobenchmark: nginx

For nginx HTTP benchmark, we conduct experiments with three workloads with static contents, namely `SpecWeb`, `LSP`, `LSP/2`. For all workloads, we configured nginx to serve files from memory rather than disks, to avoid disks to be a bottleneck.

SpecWeb: We test the same HTTP workload used in [29]. In this workload, each client connection initiates six HTTP requests. The content size ranges from 30 to 5,670 bytes (704B in average), which is adopted from the static file set of SpecWeb 2009 Support Workload [14].

[4]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.

LSP: We used the data collected from the CDNs of a large service provider. In this workload, the number of HTTP requests per connection ranges between 1 and $1,597$ (2.3 in average). Each content size ranges between 1 byte and 265 MB (12.7 KB in average). The files larger than 60KB contribute roughly 50% of total traffic.

LSP/2: Due to the large object size of `LSP` MegaPipe with only five cores fully saturates two 10G links we used. For the `LSP/2` workload, we change the size of all files by half (so 12.3to observe the multi-core scalability behavior more clearly.

We present the measurement result in Figure 7 for each workload. For all three workloads, MegaPipe significant improves the performance for both single-core and multi-core cases. MegaPipe with `LSP/2`, for instance, improves the performance by 40% (single core) and 68% (eight cores), with a better parallel speedup (from 5.4 to 6.4) with eight cores.

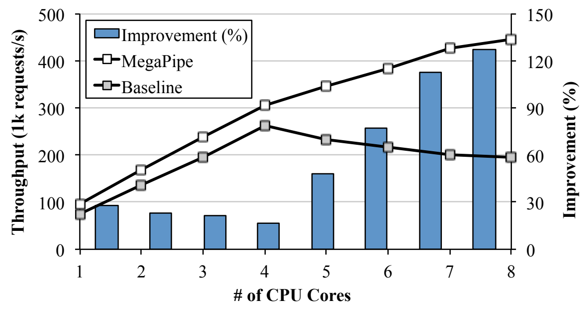
6 Related Work

In this section, we explore the similarities and differences between MegaPipe and existing work.

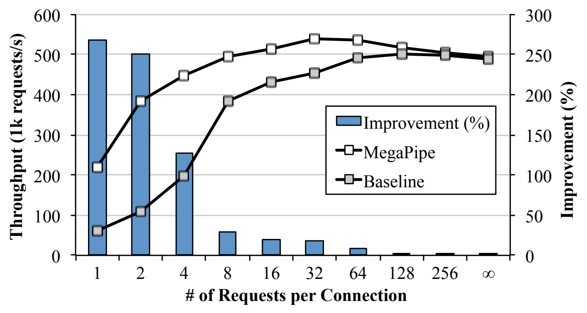
Other Asynchronous I/O Frameworks: Like MegaPipe, the Event Completion Framework [1] and Lazy Asynchronous I/O [23] support a programming model based on event completion notifications, rather than readiness. They do not address the performance and scaling limitations that MegaPipe does.

POSIX AIO defines functions for asynchronous I/O in UNIX [16]. Interestingly, it also supports a form of I/O batching: `lio_listio()` for AIO commands and `aio_suspend()` for their completion notifications. POSIX AIO is not particularly designed for sockets, but rather, general files. For instance, it does not have an equivalent to `accept()` or `shutdown()`.

System Call Batching: While MegaPipe's batching was inspired by FlexSC [30, 31], the main focus of MegaPipe is network 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; 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 connection is isolated to a single core.

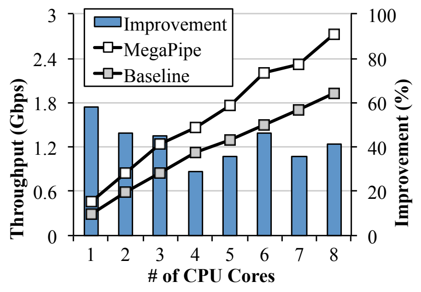


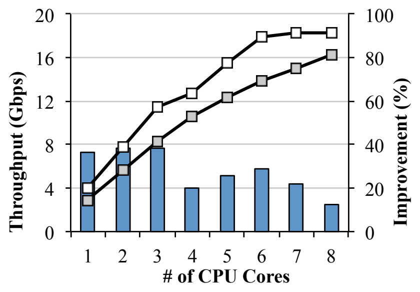
(a) Throughput comparison for a fixed connection length of 4 requests per connection.



(b) Throughput comparison for an 8 core server and a varying number of requests per connection. Note that the x-axis is a log2 scale, and that infinite connection length indicates that connections were not closed and reconnected during the test.

Figure 6: Comparison of memcached throughput for a server and two clients, each with 512 concurrent connections.





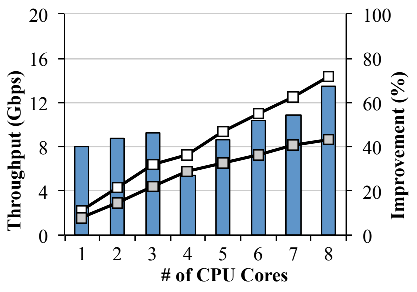


Figure 7: Evaluation of nginx throughput for the SpecWeb, LSP, and LSP/2 workloads.

Kernel-Level Network Applications: Some network applications are partly implemented in the kernel, tightly coupling performance-critical sections to the TCP/IP stack [9, 20, 24, 26]. While this improves performance, it comes at a price of limited security, programmability, and portability. MegaPipe instead retains the benefits of user-level programming.

Multi-core Scalability: Past research has shown that partitioning CPU cores is critical for linear scalability of network I/O on multi-core systems [22,29,32,34]. The main ideas are to maintain flow affinity and minimize unnecessary sharing between cores. In Section 3.4.2, we addressed the similarities and differences between Affinity-Accept [29] and MegaPipe. In [22], the authors address the scalability issues in VFS, namely inode and dentry, in the general context. We discusses in Section 3.4.1 that the VFS overhead can be completely eliminated in most cases, for network sockets.

The Chronos [25] work also proposes direct coupling between multi-queue NICs and applications for maximum performance. While MegaPipe has focused on increased throughput, Chronos aims to minimize latency. Unlike MegaPipe, Chronos removes the kernel from connection handling, exposing NIC queues directly to user-space memory. While this does avoid in-kernel scalability issues, it also loses the generality of TCP connection handling which is traditionally provided by the kernel.

7 Conclusion

Message-oriented network workloads, where connections are short and/or message sizes are small, are CPU-intensive, especially on multi-core systems. In this paper, we introduced MegaPipe, a new programming interface for high-performance I/O. MegaPipe exploits many performance optimization opportunities that used to be hindered from the existing network API semantics, while being still simple and easily applicable existing event-driven servers. Evaluation with the microbenchmark, memcached, and nginx showed significant improvement, in terms of both single-core performance and its parallel speedup on a 8-core system.

References

[1] Event Completion Framework for Solaris.

http://developers.sun.com/solaris/articles/event_completion.html.

[2] GNU Portable Threads. http://www.gnu.org/s/pth/.

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

[4] Kernel Asynchronous I/O (AIO) Support for Linux. http://lse.sourceforge.net/io/aio.html.

[5] Kqueue: A generic and scalable event notification facility. http://people.freebsd.org/~jlemon/papers/kqueue.pdf.

[6] libevent. http://www.monkey.org/~provos/libevent.

[7] memaslap - a load generation and benchmark tool for memcached servers. http://docs.libmemcached.org/memaslap.html.

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

[9] Microsoft IIS Web Server. http://www.iis.net/.

[10] nginx. http://www.nginx.org/.

[11] Receive-Side Scaling Enhancements in Windows Server 2008. http://www.microsoft.com/whdc/device/network/ndis_rss.mspx.

[12] RFS: Receive Flow Steering. http://lwn.net/Articles/381955/.

[13] RPS: Receive Packet Steering. http://lwn.net/Articles/361440/.

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

[15] The Open Group Base Specifications Issue 7. http://pubs.opengroup.org/onlinepubs/9699919799/.

[16] POSIX AIO standard. http://pubs.opengroup.org/onlinepubs/009695399/functions/xsh_chap02_08.html, 2004.

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

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

[19] sysstat: Performance Monitoring Tools for Linux. http://sebastien.godard.pagesperso-orange.fr/, 2011.

[20] BAR, M. Kernel korner: khttpd, a kernel-based web server. *Linux Journal 2000*, 76es (2000), 21.

[21] BEHREN, R. V., CONDIT, J., ZHOU, F., NECULA, G. C., AND BREWER, E. Capriccio: Scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (2003).

[22] BOYD-WICKIZER, S., CLEMENTS, A., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)* (2010).

[23] ELMELEEGY, K., CHANDA, A., COX, A. L., AND ZWAENEPOEL, W. Lazy asynchronous i/o for event-driven servers. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (2004).

[24] JOUBERT, P., KING, R., NEVES, R., RUSSINOVICH, M., TRACEY, J., ET AL. High-performance memory-based web servers: Kernel and user-space performance. In *Proceedings of the USENIX Annual Technical Conference* (2001).

[25] KAPOOR, R., PORTER, G., TEWARI, M., VOELKER, G. M., AND AMIN, V. Reducing datacenter application latency with endhost nic support. Tech. Rep. CS2012-0977, University of California, San Diego, April 2012.

[26] LEVER, C., ALLIANCE, S., AND MOLLOY, S. An analysis of the tux web server. *Ann Arbor 1001* (2000), 48103–4943.

[27] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G., ROSENBLUM, M., ET AL. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review 43*, 4 (2010), 92–105.

[28] PAI, V., DRUSCHEL, P., AND ZWAENEPOEL, W. Flash: An efficient and portable web server. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (1999).

[29] PESTEREV, A., STRAUSS, J., ZELDOVICH, N., AND MORRIS, R. T. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012), EuroSys '12, ACM.

[30] SOARES, L., AND STUMM, M. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (2010).

[31] SOARES, L., AND STUMM, M. Exception-less system calls for event-driven servers. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference* (2011).

[32] VEAL, B., AND FOONG, A. Performance scalability of a multi-core web server. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems* (2007).

[33] WELSH, M., CULLER, D., AND BREWER, E. Seda: an architecture for well-conditioned, scalable internet services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles* (2001), SOSP '01.

[34] WICKIZER, S. B., 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 *Proceedings of the 8th USENIX conference on operating systems design and implementation* (2008).