On Sockets and System Calls
Minimizing Context Switches for the Socket API

Tomas Hruby Teodor Crivat Herbert Bos Andrew S. Tanenbaum
The Network Institute, VU University Amsterdam
{*thruby,tct400,herbertb,ast*}*@few.vu.nl*

Abstract

Traditionally, applications use sockets to access the network. The socket API is well understood and simple to use. However, its simplicity has also limited its efficiency in existing implementations. Specifically, the socket API requires the application to execute many system calls like `select`, `accept`, `read`, and `write`. Each of these calls crosses the protection boundary between user space and the operating system, which is expensive. Moreover, the system calls themselves were not designed for high concurrency and have become bottlenecks in modern systems where processing simultaneous tasks is key to performance. We show that we can retain the original socket API without the current limitations. Specifically, our sockets almost completely avoid system calls on the "fast path". We show that our design eliminates up to 99% of the system calls under high load. Perhaps more tellingly, we used our sockets to boost NewtOS, a microkernel-based multiserver system, so that the performance of its network I/O approaches, and sometimes surpasses, the performance of the highly-optimized Linux network stack.

1 Introduction

The BSD socket API is the de facto standard for accessing the network. Today, all popular general-purpose systems provide a set of system calls for implementing sockets. However, sockets were invented over 30 years ago and they were not designed with high performance and concurrency in mind. Even on systems like Linux and FreeBSD, which have optimized their network stacks to the extreme, the overall network performance is crippled by the slow BSD socket API on top of it. The core problem of the socket API is that every operation requires a system call. Besides the direct cost of the trap into the operating system and back, each system call gums up the caches, the CPU pipelines, the branch predictors, and the TLBs. For some calls (like `bind` or `listen`), this is not a problem, because they occur only once per server socket. Calls like `read`, `write`, `accept`, `close`, and `select`, on the other hand, occur at very high rates in busy servers—severely limiting the performance of the overall network stack.

A clean way to remove the bottleneck is simply to redesign the API. Many projects improve the network processing speeds by introducing custom APIs [8, 14, 35]. Megapipe [20] also deliberately takes a clean-slate approach, because the generality of the existing API limits the extent to which it can be optimized for performance. In addition, it offers a fallback to a slower but backward compatible implementation for legacy software. The obvious drawback of all these approaches is that the new APIs are not compatible with the widely adopted BSD sockets and thus require software rewrites to make use of them. In addition, custom APIs typically look different on different operating systems and are frequently tailored to specific application domains.

In this paper, we investigate to what extent we can speed up *traditional* sockets. We do so by removing the worst bottlenecks, system calls, from the socket's 'data path': the time between the `socket` / `accept` and `close` system calls—during which the socket is actively used. The potential savings are huge. For instance, "system" calls that we resolve in user space are 3.5 times faster than equivalent calls in Linux. In NewtOS, removing the system calls improves the performance of lighttpd and memcached by $2\times$ to $10\times$. Instead of system calls, the socket API relies on a user space library to implement performance-critical socket functions. We do keep the system calls for less frequent operations like `bind` and `listen`, as they do not influence performance much.

Note that removing the system calls from the socket implementation is difficult. For instance, besides `send` and `recv`, we also need to move complex calls like `select` and `accept` out of the operating system. We are not aware of any other system that can do this.

As a result, our design helps scalability in speed, com-

patibility with legacy code, and portability between platforms. To the best of our knowledge, our socket design supports the networking requirements of every existing UNIX application. Moreover, without changing a single line of code, we speed up the network performance of applications like `lighttpd` and `memcached` to a level that is similar to, and sometimes better than, that of Linux—even though we run them on a slower microkernel-based multiserver operating system. For instance, we support up to 45,000 requests/s for small (20B–11kB) files on a single thread of `lighttpd` (where the application and the protocol stack run on separate hardware threads).

The key to our solution is that we expose the socket buffers directly to the user applications. Doing so has several advantages. For instance, the user process places the data directly where the OS expects them so there is no need for expensive copying *across* address spaces. Moreover, applications can check directly whether a `send` or `recv` would block (due to lack of space or lack of data, respectively). Again, the interesting point of our new design is that applications can do all this while retaining the familiar socket API and without any system calls.

Our solution is generic and applies to both monolithic and multiserver OS designs with one important condition: it should be possible to run the network stack on cores or hardware threads that are different from those used by the applications. Fortunately, this is increasingly the case. Because running the OS and the applications on different cores is good for concurrency, such configurations are now possible on some monolithic systems like Linux [34] and AIX [33], and multiserver systems like NewtOS [22].

Monolithic systems. Many monolithic systems uniformly occupy all the cores, with the execution of system code interleaving the execution of the applications. However, FlexSC [34] demonstrated that it is advantageous for systems like Linux to separate the execution of applications and operating system between different cores, and to implement system calls without exceptions or traps. Rather than trap, the application writes the system call information on a page that it shares with the kernel. The kernel asynchronously picks up these requests, executes the requested system call and ships the results back. This decouples execution of the applications and system calls.

Multiserver systems. There is even more to gain for microkernel-based multiserver systems. In the multicore era, such systems are becoming more popular and new microkernels like Barrelfish [11] emerge. Multiserver systems follow highly modular designs to reduce complexity and facilitate crash recovery [22] and "hot swapping" [12, 19]. Long thought to be unbearably slow, modern multiserver systems benefit from the availability of multicore hardware to overcome some of their historical performance issues [21, 22]. Multiserver systems consist of multiple unprivileged server (or device driver) processes which run on top of a microkernel. A single system call on a multiserver system may lead to many messages between the different servers involved in handling the call, and hence significant overhead. On the other hand, it is easy to spread multiserver systems across multiple cores so that performance critical tasks run on dedicated cores, independent of the applications. Since spatial separation of the OS and the applications is exactly what our socket implementation requires and system calls are particularly expensive, multiserver systems are a good target for sockets without system calls.

Contributions The contributions of our work are:

1. We show a new design of BSD network sockets in which we implement most network operations without any system calls.
2. We show how this design allows applications to run undisturbed while the network stack works on their requests asynchronously and concurrently.
3. We evaluate the performance of the novel design in the context of reliable multiserver systems (NewtOS [22]), using `lighttpd` web server and the `memcached` distributed memory object caching system to show that the new implementation can also bring competitive performance to systems long thought to be unbearably slow.

The rest of the paper is organized as follows. First, in Section 2 we discuss recent efforts to enhance network I/O, highlighting the weak points and describing how we address them. We present the details of our design in Section 3, 4 and 5, its implementation in Section 6 and implications for reliability in Section 7. We evaluate the design in Section 8 and we conclude in Section 9.

2 Motivation and Related Work

Although the socket API is well understood and broadly adopted, it has several performance issues. Primarily, the API was not designed with high concurrency in mind. Reading from and writing to a socket may block—suspending the entire application. To work around this limitation, applications spawn multiple processes (one for each connection), or use multiple threads [28]. Both of these approaches require switching of threads of execution with significant performance overhead, and mutual synchronization, which also affects performance and makes the software complex and error prone.

Nonblocking variants of socket operations allow handling of multiple sockets within a single thread of execution. However, probing whether a system call would succeed requires potentially many calls into the system. For

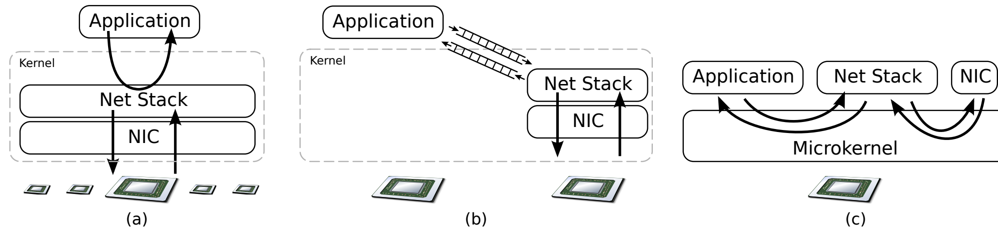

Figure 1: Net stack configurations : (a) Linux / BSD / Windows (b) FlexSC / IsoStack (c) Multiserver system.

instance, a nonblocking `read` is likely to return `EAGAIN` frequently and consume resources unnecessarily.

To avoid needless system entries, system calls like `select`, `poll`, and their better optimized variants like `epoll` or `kqueue` [24] let the application ask the system whether and when it is possible to carry out a set of operations successfully. Although `select` greatly reduces the number of system calls, the application needs to use it first to query the system which of the potential writes and reads the system will accept. This still leaves many system calls which cross the protection boundary between the applications and the system.

Besides using `select` and friends, developers may improve efficiency by means of asynchronous I/O. Such operations initially communicate to the operating system the send or receive requests to execute, but do not wait for the operation to complete. Instead, the application continues processing and collects the results later. The POSIX asynchronous I/O calls provide such `send`, `recv`, `read` and `write` operations, but even in this case, the execution of the application is disrupted by the system calls to initiate the requests and to query their status.

Monolithic systems like Linux implement system calls by means of exceptions or traps which transfer the execution from the application to the kernel of the operating system (Figure 1a). The problem of this mode switch and its effect on the execution of the application has been studied by Soares et al. in FlexSC [34]. They demonstrated that if the system runs on a different core, it can keep its caches and other CPU structures warm and process the applications' requests more efficiently in terms of the instructions-per-cycle ratio. Multithreaded applications like Bind, Apache and MySQL can issue a request and instead of switching to the kernel, they can keep working while the kernel processes the requests.

Although FlexSC is a generic way to avoid exceptions when issuing system calls, single threaded high performance servers can take little advantage without modification of their code using the *libflexsc* library [35]. Event-driven servers based on the *libevent* library [4] like nginx [10] and memcached [9] need only modest changes, however modification of other servers like lighttpd [5] would require more work. In addition, the kernel still needs to map in the user memory to copy between the application's address space and its own.

IsoStack [33] does not offer a generic solution to the system call problem and focuses solely on the AIX network stack. In particular, it reduces contention on data structures and the pollution of CPU structures that results from the interleaved execution of the network stack and applications by running the stack on separate core(s) (Figure 1b). Since the stack has its own core, it can poll the applications for data. To pass commands to the IsoStack, applications need to use the kernel—to access per-core notification queues that are shared by all applications on the core. Although each socket has its own command and status queues, there is only one notification queue per core, so the number of queues to poll is limited.

MegaPipe [20] features per-core queues for commands and completions. Unlike IsoStack, the applications do not share those queues. Specifically, each application opens its own private queue for each core on which it runs. Since the applications using MegaPipe do not run side-by-side with the network stack but on top of the kernel, the stack cannot poll those queues, but MegaPipe reduces the overhead of system calls by batching them. Making one kernel call for multiple system calls amortizes the overhead, and the batching is hidden by a higher level API. However, the API differs significantly from POSIX. The authors explicitly opted for a clean-slate approach, because the generality of the existing socket API "limits the extent to which it can be optimized in general".

Windows introduced a similar mechanism in version 8, the so-called registered I/O, also known as "RIO" [8]. The internals of the communication between user space and the kernel in Windows is much more exposed. Programmers need to create a number of queues tailored to the problem they are solving, open the sockets independently and associate them with the queues. One significant difference is that RIO explicitly addresses the optimization of data transfer. Specifically, the user needs to preregister the buffers that the application will use so the system can lock the memory to speed up the copying.

Microkernel-based multiserver systems implement system calls by means of message passing. Typically, the application first switches to the kernel which then delivers the message to the process responsible for handling the call and, eventually, sends back a reply. This makes the

system call much more costly than in a monolithic system and more disruptive. It involves not only the application and the kernel, but also one or more additional processes (servers), as shown in Figure 1c. For instance, the first column in Table 1 shows the cost in cycles of a `recvfrom` call that immediately returns (no data available) in NewtOS if the kernel is involved in every IPC message and the system call traverses three processes on different cores—a typical solution that is, unfortunately, approximately 170 times slower than the equivalent call in Linux.

One of the problems of such an implementation is that the network stack asks the kernel to copy the data between the application process and itself. Compared to monolithic systems, remapping user memory into the address space of the stack is much more expensive, as the memory manager is typically another independent process. To speed up the communication, we have proposed to convert the messaging into an exception-less mechanism when the system uses multiple cores [22]. Specifically, the user process performs a single system call, but the servers themselves communicate over fast shared-memory channels in user space. As shown in the second column of Table 1, doing so speeds up the `recvfrom` operation by a factor of 4.

However, if we avoid the system call altogether and perform the entire operation in (the network library of) the user process, we are able to reduce the cost of the `recvfrom` to no more than 137 cycles—150 times better than the fast channel based multiserver systems, and even 3.5 times faster than Linux.

Rizzo et al. use their experience with netmap [30] to reduce overheads for networking in virtual machines [31], in which case it is important to reduce the amount of VM exits, data allocation and copying. While they work on the network device layer, the VM exits present similar overhead as the system calls for sockets.

Marinos et al. [25] argue for user space network stacks that are tightly coupled with applications and specialized for their workload patterns. The applications do not need to use system calls to talk to the stack and do not need to copy data as the software layers are linked together. The stack uses netmap for lightweight access to the network cards. Similarly, mTCP [23] argues for user space network stack since decoupling the application and the heavy processing in the Linux kernel (70-80% of CPU time) prohibits network processing optimizations in the application. These setups trade performance for loss of generality, portability and interoperability (the network stacks are isolated within their applications) and usually monopolize network interfaces.

Solarflare's OpenOnload [7] project allows applications to run their own network stack in user space and bypass the system's kernel. While it provides a low-latency POSIX interface, it works only with Solarflare NICs as it needs rich virtualization and filtering hardware features.

Kernel msgs	User space msgs	No system calls	Linux
79800	19950	137	478

Table 1: Cycles to complete a nonblocking `recvfrom()`

Our design draws inspiration from all the projects discussed above. As none of them solves the problem of marrying reliable systems to high-performance using the existing socket API, we opt for a new implementation that eleminates the system calls during the active phase of a socket's lifetime (between socket creation and `close`).

3 Sockets without System Calls

The socket API consists of a handful of functions. Besides the socket management routines like `socket`, `close`, `connect`, `set/getsockopt` or `fcntl`, applications can only read/receive from and write/send to the sockets, although there are different calls to do so. A socket can act as a stream of data or operate in a packet-oriented fashion. Either way, the application always either sends/writes a chunk of data (possibly with attached metadata), or receives/reads a chunk of data. We limit our discussion to general reads and writes in the remainder of this paper, but stress that the arguments apply to the other calls too. To help the reader, we will explicitly mention what parts of NewtOS are similar to existing systems. The main point of our work is of course entirely novel: high-speed BSD sockets that work with existing application and without system calls in the active phase, designed for reliable multiserver systems .

The focus of our design is to avoid any system calls for writing and reading data when the application is under high load. We achieve this by exposing the socket buffers to the application. When reading and writing sockets using legacy system calls, applications cannot check whether or not the call will succeed, before they make it. To remedy this, we allow the applications to peek into the buffer without involving the operating system at all. Thus, when the application sees that the operating system has enough space in the socket buffer, it can place the data in the buffer right away. Similarly, when it sees that the buffer for incoming data is not empty, it fetches the data immediately.

In summary, our key design points are:

1. The socket-related calls are handled by the *C library* which either makes a corresponding system call (for slow-path operations like `socket`, or if the load is low), or implements the call directly in user space.

2. The API offers standard BSD sockets, fully compatible with existing applications.

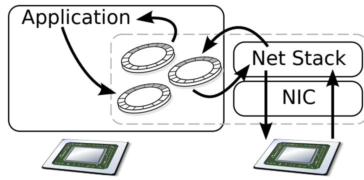

Figure 2: Exposed socket buffers – no need to cross the protection boundary (dashed) to access the socket buffers.

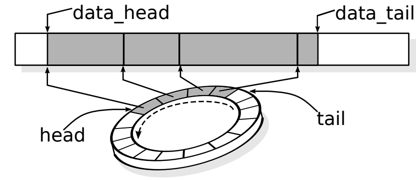

Figure 3: A socket buffer – ring queue and data buffer. White and shaded present free and used / allocated space.

3. We use a pair of queues per socket and the queues also carry data. Each queue is the socket buffer itself.
4. The application polls the sockets when it is busy and wakes up the stack by a memory write to make sure the stack attends to its requests very quickly.
5. The network stack polls on per-process basis (rather than each socket individually).
6. We avoid exception-based system calls without any batching, and the application only issues a blocking call when there is no activity.

Although we discuss the design of the buffers and the calls separately in Sections 4 and 5, they are closely related. For instance, we will see that exposing the buffers to the application allows a reduction in the number of system calls, as applications can check directly whether or not a read or write operation would fail.

4 Socket Buffers

A socket buffer is a piece of memory the system uses to hold the data before it can safely transmit it and, in the case of connection oriented protocols, until it is certain that the data safely reached the other end. The receive buffer stores the data before the application is ready to use it. Our socket buffers are different from traditional ones in that they are premapped and directly accessible by the application as presented in Figure 2. We exposed the socket buffers to user space much like netmap [30] exposes buffers of network interfaces and it is similar to how FBufs [15], Streamline [14] or IO Lite [29] map data buffers throughout the system for fast crossing of isolation domains.

4.1 Exposing the Buffers

The system allocates the memory for the socket and maps it into the address space of the application right after socket creation. The advantage is that the system does not need to create the mappings when it transfers the data between its address space and the application. This is similar to Windows RIO. However, unlike RIO, the programmers do not need to register the buffers themselves. Instead, buffers are ready at the time the socket opens. This keeps the use of sockets simple.

The socket buffers are split in two parts—one for incoming and one for outgoing data. Each of these parts is further split into an unstructured memory area for data and an area that forms a ring queue of data allocation descriptors, as presented in Figure 3. Both the application and the system use the queues to inform the other side about the data they have placed in the socket. This is similar to IsoStack's [33] per socket data queues. However, we do not require notification queues.

Mapping the buffers entirely in the address space of the application makes the data path (e.g., reads and writes) cheaper and the control path (e.g., the `socket` call) more expensive as the mapping takes place when creating the socket. Thus, it is a potential performance bottleneck for servers which open and close many connections. In RIO, the mappings are initiated by the application after it opens the socket and only then the application associates them with the socket. In contrast, we amortize the cost of mapping the buffers by reusing the same mappings (as we describe in Section 5.5), which is trivially possible because the operating system creates them.

With traditional "hidden-buffer" sockets, the application lacks feedback on how quickly the network stack can drain the buffers when sending. This contributes to the so-called Buffer Bloat [17] problem. In general, it means that applications blindly send as much data as they are allowed, hoping that the network will somehow deliver them. As a result, too much data accumulates in network stacks, staying too long in large buffers before eventually being dropped and retransmitted. Linux partially fixes the problem by employing an algorithm called Byte Queue Limits (BQL) [6], which limits the number of bytes the system can put in the buffers of the network interfaces. Instead of blindly pushing as much data as it can, it checks how much data the interface sent out during a period of time and sets this as a limit for itself. Unfortunately, the process is completely hidden from the application. For instance a video streaming server could decrease the video quality to reduce the bitrate if it knew that the connection cannot transmit fast enough. The exposed socket buffers' head and tail pointers provide enough information for a BQL-like algorithm in the library to limit the number of

bytes the applications write in the socket (e.g., by failing the `write` call). POSIX provides `ioctl` calls to query such information, however, our sockets can do it cheaply, without the overhead of system calls.

4.2 Socket Buffers in Practice

The ring buffer part of the socket has 64 slots in our prototype and the application can change the size of the buffer and also change the number of slots of each of the ring buffers (using `setsockopt`) as the needs of each application can differ. For instance, senders with many small writes or receivers with mostly small requests may prefer higher slot counts with a smaller area for data.

Using single-producer, single-consumer ring buffers (or queues) is convenient as they are lock-free [18] and detecting whether they are empty or full is achieved inexpensively by examining the head and tail pointers. Likewise, it is simple to indicate activity by incrementing the head or tail pointer. Due to the pipe-like socket nature, we can allocate the data sequentially also, and deallocate them in the same order, so there is no fragmentation and testing for full or empty is equally cheap as for the descriptor ring.

Since the ring queues are in shared memory and the operating system cannot trust the application, it uses memory protection for the ring structure so that the receiver cannot change the values that the system later uses for deallocating the data. Only the head pointer is writable by the consumer—as it must modify it to indicate progress. We place the head pointer in a separate memory page to enforce the protection.

Both the application and the network stack must free the buffers when the consumer has processed the data. The C library, which implements POSIX, tries to reclaim buffers after we successfully completed a write to a socket, keeping the socket free for new writes and avoiding blocking and system calls or when allocation fails. Of course, if the socket buffer is full and the operation is blocking, the application must block as we describe in Section 5.4.

The POSIX system call API does not guarantee correctness of execution when multiple threads or processes use the same socket. However, it provides a certain degree of consistency and mutual exclusion as the operating system internally safeguards its data structures.

For example, many simultaneous calls to `accept` result in only one thread acquiring a socket with the new connection. Although programmers avoid such practices, above all for performance reasons (e.g., Facebook uses different UDP ports for different memcached threads [27]), our library uses a per-socket spinlock to provide the same protection. In a well structured program, there is no contention on this lock. However, when a process forks or sends a socket descriptor through a pipe, we cannot guarantee that different processes do not cause each other harm. Therefore, the system catches such situations and transparently reverts to slow but fully compliant sockets. For this reason, the performance of legacy servers like apache or opensshd does not improve—nor does it worsen. In contrast, the vast majority of client software like `wget`, `ftp` or web browsers, and modern event-driven servers like `lighttpd`, `nginx` and `memcached`, do not need any change to take full advantage of the new sockets.

5 Calls and Controls

In this section we describe how the network stack and the applications communicate with each other, how we implement the system calls and signaling without involving the kernel.

5.1 Notifying the System

When an application writes to a socket, it needs to tell the system to handle the request. When the application runs "on top of" an operating system (i.e., the system's kernel runs only upon requests from user processes, or when an external event occurs), the only practical means for an application to tell the system about its request is a system call. Although it is possible to have a system thread which periodically polls the sockets, much like in VirtuOS [26], we think this is impractical. It would not only make running of the system thread dependent on the scheduler, but it would also require descheduling of the user process to switch to the polling thread, which would impact the performance of all the threads sharing the core. It also requires a good estimate for how long and how frequently to poll, which is highly workload dependent.

In contrast, we exploit the fact that the network stack runs on a different core than the applications. The stack can poll or periodically check the sockets without disrupting the execution of applications. On modern architectures, it is possible to notify a core about some activity by a mere write to a piece of shared memory. For instance, `x86` architectures offer the `MWAIT` instruction which halts a core until it detects a write to a monitored memory region, or until an interrupt wakes it up. This allows for energy-efficient polling.

Using the memory to notify across cores has several additional important advantages. First, it is cheap. Although the write modifies a cache line, which may invalidate it in the cache of other cores resulting in some stalls in the execution, this is a much smaller overhead than switching to the system, which would suffer the same caching problem, but on a much larger scale. Second, the sender of the notification can continue immediately. More importantly, the notification does not interrupt the work of the code on the receiver side. Therefore, when the network stack

is busy, it can continue its work in parallel to the applications and check the socket eventually. Interrupting its execution would only make things worse without passing any additional useful information to the network stack.

5.2 Notifying the Application

NewtOS has no need to keep notifying the applications. Applications are structured to keep processing as long as they can find a socket to read from, which our sockets allow without direct interaction with the system. Under high load, there is usually a socket with data available. When the load is low, making a system call to block the application is generally fine and it is the only way to avoid wasting resources and energy by polling. This way applications explicitly tell the system to wake them up when new work arrives. In the rare case that the application domain demands extremely low latency and cares less about resource usage and energy consumption, the sockets can switch to active polling instead.

Traditionally, applications use `select`-like system calls to suspend their execution, as they cannot efficiently decide whether an operation would block. We still offer `select` for legacy applications, but as we will see in Section 5.4 our implementation requires no trap to the operating system if the load is high.

5.3 Socket Management Calls

Except for extremely short connections, reads and writes make up the majority of the socket-related calls during a socket's lifetime. Nevertheless, busy servers also accept and close connections at a very high rate. Since our sockets lack a command queue from the application to the operating system, it appears, at first glance, that we cannot avoid a high rate of system calls to create and manage the sockets. In reality, we will show that we can handle most of them in user space. Specifically, while we still implement the `socket` call as a true system call, we largely avoid other calls, like `accept` and `close`.

Fortunately, the `socket` call itself is not used frequently. For instance, the server applications which use TCP as the primary communication protocol use the `socket` call to create the listening server socket, setting up all data-carrying sockets by means of `accept`. In client applications the number of connections is limited to begin with, and the `socket` call is typically not used at high rates. Phrased differently, a slightly higher overhead of creating the connection's socket is acceptable.

In general, we can remove the system calls only from those API functions that the application either uses to collect information from the system, or that it can fire and forget. As all other calls may fail, the application needs to know their result before continuing. We implement the management calls in the following way:

accept As mentioned earlier, the operating system premaps the buffers for a pending connection in the listening socket's backlog—before the system announces its existence to the application. As a result, the `accept` itself can be handled without a system call. The application reads the mapping information from the listening socket and writes back the acknowledgment that it accepted the connection.

close Closing a connection always succeeds, unless the file descriptor is invalid. After checking the validity of the descriptor, the library injects a close data descriptor, returns success to the application and forgets about the call. The system carries it out once it has drained all data from the socket, asynchronously with the execution of the application. It also unmaps the socket from the application's address space. In case `SO_LINGER` is set, we check whether the buffer is not empty in which case we must block and a system call is acceptable.

listen, bind These calls are infrequent as they are needed only once for a server socket. For performance, it makes no difference whether they make system calls.

connect Connecting to a remote machine is slow and the overhead of the system call is acceptable. We implement connect on a nonblocking socket by writing a "connect" descriptor into the socket buffer and we collect the result later when the system replies by placing an error code in the socket buffer.

In the remaining subsections, we discuss the control calls that we execute in user space (`select`, `accept`, and `close`) in more detail.

5.4 Select in User Space

Our sockets enable the application to use cheap nonblocking operations to poll the buffers to check whether it is possible to read or write. However, many legacy applications perform this check by issuing `select` or a similar call. In this case, our library select routine sweeps through all the sockets indicated by the file descriptor sets supplied and returns immediately if it discovers that any of the sockets would accept the desired operation. Only if there is no such socket, it issues a call to the system, passing along the original descriptor sets. A blocking call internally checks whether it can complete or whether it needs to block using `select`, however, without the overhead of copying an entire set of file descriptors.

It is possible to optimize the polling further. For instance, the application can keep polling the sockets longer

before deciding to make the system call. The polling algorithm may also adapt the polling time based on the recent history, however, this is beyond the scope of this paper.

We implemented one simple optimization. Since the head and tail pointers of the ring buffers reside in memory shared between different cores, reading them often may result in cache line bouncing as another core may be updating them. We do not always need to read the pointers. After we carried out the last operation, we make a note whether the same operation would succeed again, i.e. whether there is still space or data in the buffer. As we complete the operation, the values of the pointers are in local variables assigned to registers or local cache. Taking the note preserves the information in non-shared memory. In addition, we can condense the information into a bitmap so we can access it in `select` for all the sockets together, thus not polluting the local cache unnecessarily.

Although it is not part of POSIX and hence not portable, we also implemented a version of `epoll` as it is more efficient than pure `select`. The library injects the file descriptors to monitor by means of writes to the socket used for epoll, and polls this socket by means of reads and a `select`-like mechanism for the pending events. In contrast to Linux and similar to our writes, applications can issue many `epoll_ctl` calls without the system call overhead. Likewise `epoll_wait` often returns the events right away in user space, much like our `select`. The benefit of `epoll` is not only because it is O(1), more importantly, it allows the library to poll fewer sockets.

5.5 Lazy Closing and Fast Accepts

The obvious bottleneck of our approach is that we must set up the mappings of the buffers before we can use the sockets. This is a time consuming operation, especially in a multiserver system because it requires a third component to do the mappings on behalf of the network stack. This is typical for multiserver systems.

We amortize some of this cost by not closing the socket completely, but keeping some of the mappings around for reuse when the application opens new ones. This is especially useful in the case of servers as they handle many simultaneous connections over time. Once a connection closes, the server accepts a new one shortly thereafter.

We let the system decide when to finalize closing a socket. It allows the network stack to unmap some of the buffers in case of memory pressure. For instance, when an application creates and closes a thousand connections and keeps the sockets around while it does not use them, the stack can easily reclaim them. Note that once the application tells the system that it has closed a socket and thus will not use it any more, the system is free to unmap its memory at any time and the application must not make any assumptions about the algorithm. If the application

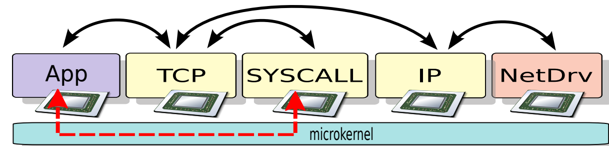

Figure 4: Architecture of the network stack

touches the socket's memory without opening/accepting a new one, the application may crash.

The default algorithm we use keeps the sockets in a "mapped" state as long as at least half of the sockets are in use. Once the number of the active sockets is smaller than the number of the closed ones, we unmap all the closed ones. The idea is that the number of connections a server application uses at a time oscillates around an average value with occasional peaks. Thus, it is good to have some preopened sockets available. However, when the number of connections drops significantly, we want to free many of them and start exploring for the new average. Of course, when the application wants more connections, the stack creates them at a higher cost and latency.

For applications, the `accept` and `close` calls are very cheap operations. On accept, the library reads a description of the new socket, but the mappings are already present. Similarly, on a close, the application writes a close descriptor into the socket, while the unmapping occurs asynchronously in the background.

6 Fast Sockets in NewtOS

We implemented the new sockets in a multiserver system, which are often found in deployments where reliability is the primary concern, for example QNX [32]. Fixing the performance limitations of multiserver systems is **(a)** challenging as the solution must not compromise the reliability and **(b)** urgently needed to allow their wider acceptance. Instead of trading performance for reliability, we use more memory (which is nowadays plentiful) and more cores (which are becoming equally abundant, e.g., Intel announced new 72-core Knights Landing chip [3]). The reliability of these systems typically consists of a modular design that facilitates properties like fault isolation, crash recovery, and live updatability.

In an extremely modular design, NewtOS' network stack itself is broken into several single-threaded servers (TCP, UDP, IP, packet filter) and device drivers. Figure 4 presents a subset of the components. The syscall server dispatches the system calls. The solid arrows represent communication without system calls and kernel involvement, while the dashed one stands for a traditional kernel IPC based system calls. NewtOS demonstrated [22] that pinning individual servers to dedicated cores removes the biggest overhead of multiserver systems—context

switching— and further increases the performance as various servers execute in parallel.

Our implementation introduces asynchrony and parallelism between the system and the applications. We let applications, which have internal concurrency, submit requests to the system asynchronously. Unlike IsoStack [33] or FlexSC [35] which require the introduction of additional (kernel) threads into the system, multiserver systems are already partitioned into isolated processes and satisfy very easily our requirement that the network stack must run on its own core(s) or hardware thread(s). One of the benefits of our design is that the OS components run independently and hardly ever use the kernel.

6.1 Polling within the Network Stack

The TCP and UDP processes of the network stack poll the sockets when the applications are busy but are idle when there is no work. However, the applications do not know when the stack is idle. Therefore they keep signaling the stack by writes to a special memory area to indicate that there is some activity in user space. The stack first needs to check where the activity comes from. The `MWAIT` instruction can monitor only a single memory location at a time, usually of a cache line size. Because all simultaneously running applications must have write access to this memory, passing specific values would compromise their isolation. It is also impractical for the stack to always sweep through all the existing sockets, as that would mean fetching at least the tail pointer of each socket (a whole cache line) into its local cache. If there were too many idle sockets, this would result in thrashing the cache while not helping the progress. For this reason, both MegaPipe [20] and IsoStack [33] ruled out per-socket command queues.

Our implementation maps one additional shared memory region in the application's address space once it creates the first network socket. This region is private to the application and has an integer-sized part at its beginning where the application indicates that it has been active since we last checked. The rest of the memory is a bitmap with one bit for each socket that belongs to the application. The stack polls the activity field of each application and only if it is positive, it inspects the bitmap. The bitmap grows and shrinks with the number of sockets the application uses concurrently. The network stack never polls any socket or application indefinitely to avoid starvation.

It is possible that the application is setting bits in the bitmap at the same moment as the stack clears them to mark the already processed sockets. To avoid possible races, we require the application to use an atomic instruction to set the bits in the bitmap. The stack processes the bitmap in chunks which it reads and clears at the same time using an atomic "swap" instruction. According to the findings in [13], x86 atomic and regular cross-core operations are similarly expensive, hence the atomicity requirement does not constitute a major bottleneck. In case the application does not comply, the application itself may suffer, as the stack may miss some of its updates. However, doing so will not affect the correctness of the execution of the stack, or other applications.

6.2 Reducing TX Copy Overhead

When an application sends data (writes to a socket), the system must copy them into its own buffers so it can let the application continue. Otherwise the application would need to wait until the data were transmitted from the supplied buffer as it is free to reuse the buffer once it returns from the `write`. In a monolithic system like Linux the kernel must do the copy. In multiserver systems, it is typically done by the kernel too, since only the kernel can access all memory and transfer data between protection domains. Asking the kernel to copy the data unnecessarily disrupts the execution of other applications and servers due to the contention on the kernel as microkernels usually have one kernel lock only.

Therefore we use shared memory to transfer the data between the address space of the application and the network stack. Similar to Isostack [33], by exposing the DMA ready socket buffers to the application, the application can directly place the data where the network stack needs it so it does not have to touch the payload at all. The last and only copy within the system is the DMA by the network interface. On the other hand, since the POSIX API prescribes copy semantics, this means that the application must do the copy instead of the system. Doing so is cheaper than constantly remapping the buffers in a different address space. In addition, since the applications do not share their cores with the system, the copy overhead is distributed among them, relieving the cores which host the servers of the network stack. This is extremely important as, for the sake of simplicity for reliability, the TCP and UDP servers in NewtOS are single threaded and do not scale beyond using a single core each.

Unfortunately, it is not possible to remove copying within the TCP and UDP servers when receiving data as the hardware would need more complex filtering logic (e.g, Solarflare [7]) to be able to place the data in the buffers of the right socket. However, the copy is within the same address spaces, not between protection domains.

7 Implications for Reliability

Multiserver systems are touted for their superior reliability and are capable of surviving crashes of their servers. Clearly, our fast socket implementation should not lower the bar and trade reliability back for performance. Shared

memory in particular is a source of potential problems for reliability, especially if one thread or process may corrupt the data structures of another one. We now explain why our sockets are safe from such problems.

First, note that the new sockets implementation uses shared memory in a very restricted way—to exchange data. The only data structure we share is a ring queue of descriptors. As this data structure is writable only by its producer, while the consumer cannot modify it, it is not possible for consumers to harm the producers.

Of course, our most important requirement is that applications cannot damage the network stack and compromise the system. Since the queue has a well defined format and only contains information about sizes and offsets within the data part of the mapped socket buffer, the consumer is always able to verify whether this information points to data within the bounds of the data area. If not, it can be ignored or reported. For instance, the network stack can ask the memory manager to generate a segmentation fault for the application. Even if the application is buggy and generates overlapping data chunks in the data area, or points to uninitialized or stale data, it does not compromise the network stack. Although POSIX copy semantics, as implemented by commodity systems, prohibits applications changing data under the system's hands, the fact that our sockets allow it is no different from the applications generating garbage data in the first place.

The only serious threat to the network stack is that the application does not increment the head pointer of a queue properly or does not stop producing when the queue is full. Again, this may only result in the transmission of garbage data or in not transmitting some data. This behavior is not the result of a compromised system, but of an error within the application.

Similarly, an application can keep poking the stack and waking it up without submitting any writes to its sockets. This is indeed a waste of resources; however, it does not differ from an application keeping the system busy by continuously making a wrong system call.

8 Evaluation

We evaluated our design in our system using a 12 core AMD Opteron Processor 6168 (1.9 GHz) with a 10G Intel i82599 card and we compared it to Linux 3.7.1 running on the same machine. We ran the benchmarks on a dual-socket quad-core Intel Xeon 2.26 GHz (E5520) running Linux 3.6.6 connected with the same network card.

For a fair evaluation of the system call overhead, we present a performance comparison of our network stack with sockets that use our new design and those that use system calls. The baseline implementation uses exactly the same code, but the stack does not poll and the applications use system calls to signal writes and to check whether reading is possible. Since we use the exposed buffers to transfer data between the stack and the application in both cases, the baseline already benefits from kernel not copying between protection domains. It makes it already significantly faster than any previous implementation of sockets in a multiserver system as reported, for example for MINIX 3 or NewtOS, in [22].

Our network stack is based on the LwIP [16] library which is highly portable but not optimized for speed. Performance-wise, it has severe limitations. For instance, all active TCP control blocks are kept in a linked list which is far from optimal when the number of connections is high. As this is neither a fundamental limitation, nor the focus of our paper, we limited the evaluation to 128 concurrent connections. The socket implementation is independent of LwIP and we can use any other library which implements the protocols. To put our results in perspective, we also compared against Linux, one of the best performing production systems. We stress that it is not an apples-to-apples comparison due to the different designs, complexity and scalability of the systems.

Although our sockets are designed for general use, their main advantage is that they avoid system calls when the applications experience high load—something that applies mostly to servers. We therefore conducted the evaluation using the popular *lighttpd* [5] and *memcached* [9], a wide-spread distributed memory object caching system.

For fair comparison with Linux, we configured lighttpd to use `write` and both application to use `select` in Linux as well as in the case of NewtOS. Using `epoll` did not make any measurable difference for our benchmarks. In addition, it shares `sendfile`'s lack of portability. We always ran a single server process/thread as scalability of our stack and the one in Linux are fundamentally different.

8.1 lighttpd

To generate the web traffic, we used *httperf* [2] which we modified to allow a fixed number of simultaneous connections. We patched lighttpd to cache files in memory to avoid interference with the storage stack (as we focus solely on the performance of the sockets),

In the first test, we used httperf to repeatedly request a trivial "Hello World" page of 20 bytes. In this test, the web server accepts a connection, reads the request, writes the HTTP header crafted by the HTTP engine and writes the tiny content which always fits in the socket buffer. That means all the writes avoid a system call. In addition, lighttpd immediately reads from the connection again as HTTP 1.1 connections are persistent by default. The additional `read` is issued speculatively as a nonblocking operation and is always resolved in the user library. Either there are available data or not. Similarly lighttpd makes

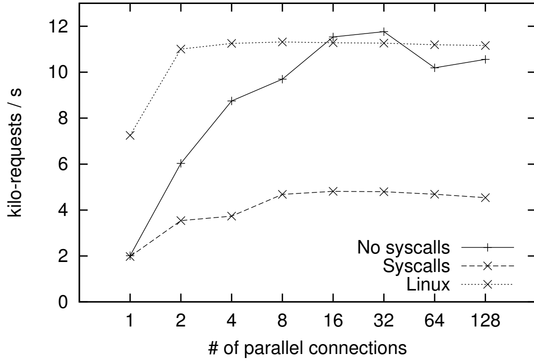

Figure 5: 1 request for a 20 B file per connection

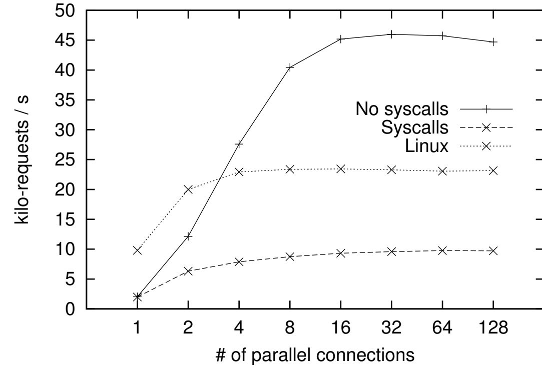

Figure 6: 10 requests-responses per connection, 20 B file

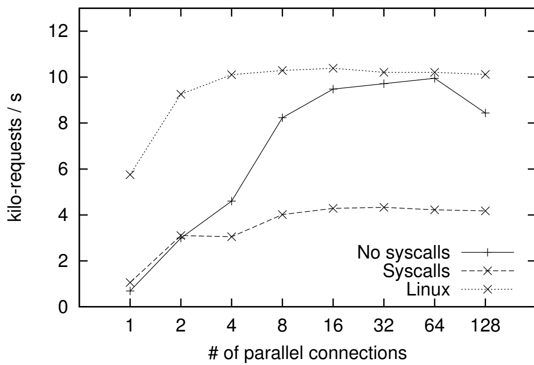

Figure 7: 1 request and a 11 kB response per connection

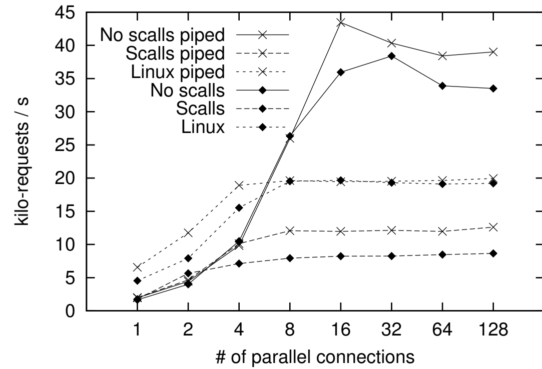

Figure 8: 10 requests for an 11 kB file

speculative `accept` calls as it always tries to accept many connections at once. As long as the number of simultaneous connections is low, every other call may just fail. This we resolve in the library as well.

Figure 5 shows that the new sockets easily outperform sockets that use system calls. In case of a single connection, the request rate is similar as the new sockets make many system calls due to low load and the latency of the connections hides the overheads for the old sockets. However, two connections are sufficient to demonstrate the advantages and the performance further scales to a level similar to Linux. We cannot compete with Linux when the number of connections is low due to higher latency within our stack (as several processes handle each packet). On the other hand, lightly loaded servers and small numbers of connections are typically not a performance problem.

In the second test, we kept each connection open for 10 request–response round trips (as servers tend to limit the number of requests per connection). Doing so removes the high latency of setting up TCP connections and increases the load on the server. Although throughput increases both for Linux and NewtOS, Figure 6 shows that in such a case the low overhead of lighttpd's speculation using the new sockets pays off and the performance quickly diverges and even significantly surpasses Linux.

According to [1], the average size of the 1000 most popular web sites (in 2012) was 1114 kB, made of 100 objects, or an average object size of approximately 11 kB [1]. In [25], 22 kB is used for evaluation as the size for static images. We use the 11 kB size in another test. Although the write size is significantly larger, the request rate is similar (Figure 7) to serving the tiny files.

In the next test we use persistent connections for 10 requests for the 11 kB objects, sent one by one or pipelined, that means sending them using a single connection without waiting. Pipelining the requests (lines with $\times$ points in Figure 8) helps Linux to avoid some reads when the load is low as several requests arrive back-to-back and the application reads them at once. It helps our new sockets even more when the load is high as we can avoid not only reads but writes as well as several replies fit in the socket.

To demonstrate how many system calls we save, we present a break down of the pipelined test in Table 2. The first column shows the percentage of all the socket calls the lighttpd makes and which are satisfied within the user space library. Even for the lowest load, we save over 90% of all system calls. 50% of `accept` calls and, more importantly, 69.48% of reads that would enter the system only to return `EAGAIN` error, fail already in user space. The remaining calls return success without entering the

	user space	select	sel / all	accept	read
1	91.20 %	99.46 %	8.84 %	50.00 %	69.48 %
2	92.79 %	81.97 %	8.79 %	48.74 %	68.36 %
4	94.97 %	62.83 %	7.99 %	47.07 %	69.35 %
8	98.16 %	33.97 %	5.39 %	44.34 %	72.04 %
16	99.93 %	4.20 %	1.59 %	32.35 %	80.80 %
32	99.99 %	2.50 %	0.27 %	8.46 %	84.27 %
64	99.98 %	9.09 %	0.12 %	3.63 %	84.43 %
128	99.97 %	17.83 %	0.14 %	4.11 %	84.34 %

Table 2: Pipelined 10× 11 kB test – percentage of calls handled in user space, fraction of selects that block, ratio of selects to all calls, accepts and reads that would block

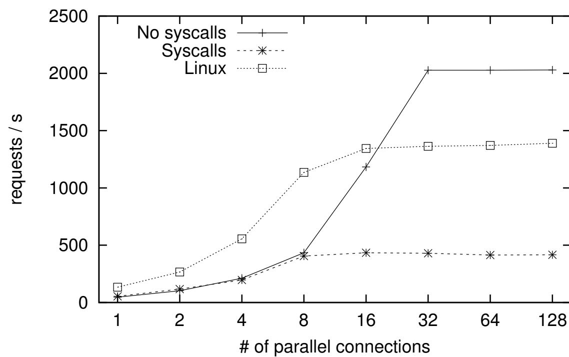

Figure 9: 1 request for a 512 kB file per connection

system at all. Since all other calls are nonblocking, only `select` can actually enter the system. Although 99% of them do so, it is less then 9% of all socket calls. As the load increases, the chance of accepting a connection is higher and the speculation pays off. On the other hand, many reads would block and we save many trips to the system and back. Interestingly, the number of `select` calls dramatically drops below 1% of all calls. Effectively, lighttpd replaces `select` by the nonblocking calls and takes full advantage of our new socket design which saves over 99% of all calls to the system. Although the fraction of `select` calls that block increases when the number of connections is higher than 32, the total ratio to other calls remains close to 0.1%. Since other calls do not block more frequently, it only indicates that they can successfully consume all available work. Although infrequently, the user process needs to block time to time.

In the last test we request one 512 kB file for each connection. This workload is much less latency sensitive as transmitting the larger amount of data takes more time. In contrast to the smaller replies, the data throughput of the application and the stack is important. Although Linux is faster when latency matters, results in Figure 10 show that minimizing the time lighttpd spends calling the

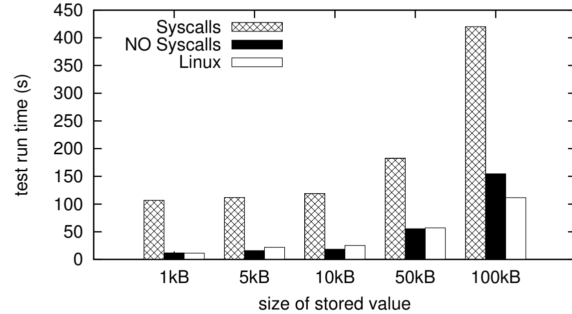

Figure 10: Memcached - test duration for different size of stored values. Test uses 128 clients. Smaller is better.

system and the overhead of processing the calls allows us to handle more than 4× as many requests as when using system calls with the same stack and 40% more requests than Linux when the load is high. Not surprisingly, the number of requests is an order of magnitude smaller than in the other tests, however, the bitrate surpasses 8 Gbps.

8.2 memcached

We use *memslap* provided by *libmemcached*, to stress the server. The test preloads different key-value pairs of the same size to a memcached server and we measure the time needed for the clients to simultaneously query 10,000 pairs each without any cache misses. For the limited space we only show a set of results for 128 client threads (Figure 9), requesting values of 1kB, 5kB, 10kB, 50kB and 100kB, respectively. The results clearly present the advantage of not using system calls in NewtOS, while it shows that the performance of this test is comparable to running memcached in Linux. In general, the benefits for memcached are similar as for the lighttpd web server.

8.3 Scalability

At first glance, scalability may be an issue with our network stack as an individual core used by the network stack may get overloaded (refer to [21] for the CPU usage of the network stack). Observe that the new sockets significantly reduce the CPU load of the TCP (and UDP) core, since copying of the data is now done by the user space library. Hence the CPU usage of TCP is similar to IP and the driver: roughly 56% (TCP), 45%(IP) and 64% (driver) for the peak load of the test presented in Figure 6. Even though the utilization is around 50%, the stack can handle up to 4 instances of the same test before it gets overloaded, peaking at 180k requests per second. Although it is counter intuitive that the cores can handle such an increase in the load, the reason is that a significant portion of the CPU usage is spent in polling and suspending when the load is low while it is used more

efficiently when the load is high and the components do processing most of the time, as we discussed in [21] as well. Especially drivers tend to have excessively high CPU usage when the load is low due to polling devices across the PCI bus.

Note that the problem of overloading a core is not directly related to the implementations of the sockets as it is possible to use them with any stack that runs on dedicated cores, for instance in a system like IsoStack or FlexSC. Nor is the network stack of NewtOS set in stone. For instance, NewtOS can also use a network stack, which combines all its parts into a single system server, similar to the network stack of MINIX 3. Doing so removes communication overheads and a multithreaded implementation of such a network stack can use multiple cores and scale in a similar way as the Linux kernel or threaded applications. Our choice to split the network stack into multiple isolated components is primarily motivated by containing errors in simpler, smaller and single threaded processes.

Most importantly, however, is that even such a network stack scales quite well as NewtOS can run multiple instances of the network stack. The new sockets make it transparent to the applications since once a socket is mapped, the application does not need to know which stack produces or consumes the data. Some architectural support from the network interfaces is needed to make sure that the right instance of the network stack handles the packets. We have built such a multistack variant of our system, and are currently evaluating it. However, the design and implementation are out of the scope of this paper and we present results that only hint at the possibilities. Specifically, with the number of cores (12) available on our test machine we ran 2 instances of the network stack and an additional fifth instance of lighttpd. This setup handles 49k requests per second more to make it the total of 229k, while the network stack is not overloaded yet. We can also use 3 instances of the single component network stack to achieve a total of 302k requests per second using 6 instances of lighttpd. The rest of the system, the driver and the syscall server use the remaining 3 cores.

9 Conclusions

We presented a novel implementation of the BSD socket API that removes most system calls and showed that it increases network performance significantly. Specifically, our sockets provide competitive performance even on multiserver systems which may have been praised by some for their modularity and reliability, but also derided because of their lack of speed. We trade performance for higher resource usage as we run the network stack on dedicated core(s) and preallocate more memory, arguing that this is justifiable given the abundance of memory and growing number of cores. We evaluated the design using *lighttpd* and *memcached* which can take full advantage of our socket design without any modification, and show that for the first time, the network performance of a reliable multiserver OS is comparable to a highly optimized production network stack like that of Linux.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Mahesh Balakrishnan for their feedback. This work has been supported by the ERC Advanced Grant 227874 and EU FP7 SysSec project.

References

[1] Average Web Page Size Triples Since 2008, 2012. `http://www.websiteoptimization.com/speed/tweak/average-web-page/`.

[2] httperf. `http://www.hpl.hp.com/research/linux/httperf/`.

[3] Intel's "knights landing" xeon phi coprocessor detailed. `http://www.anandtech.com/show/8217/intels-knights-landing-coprocessor-detailed`.

[4] libevent. `http://libevent.org`.

[5] Lighttpd Web Server. `http://www.lighttpd.net/`.

[6] Network transmit queue limits. `http://lwn.net/Articles/454390/`.

[7] OpenOnload. `http://www.openonload.org`.

[8] What's New for Windows Sockets. `http://msdn.microsoft.com/en-us/library/windows/desktop/ms740642(v=vs.85).aspx`.

[9] Distributed Caching with Memcached. `http://www.linuxjournal.com/article/7451`, Linux Journal, 2004.

[10] Nginx: the High-Performance Web Server and Reverse Proxy. `http://www.linuxjournal.com/magazine/nginx-high-performance-web-server-and-reverse-proxy`, Linux Journal, 2008.

[11] 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 *Proceedings of the Symposium on Operating Systems Principles* (2009).

[12] Baumann, A., Heiser, G., Silva, D. D., Krieger, O., Wisniewski, R. W., and Kerr, J. Providing Dynamic Update in an Operating System. In *Proc. of the USENIX Annual Technical Conf.* (2005).

[13] David, T., Guerraoui, R., and Trigonakis, V. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the Symposium on Operating Systems Principles* (2013).

[14] de Bruijn, W., Bos, H., and Bal, H. Application-Tailored I/O with Streamline. *ACM Trans. Comput. Syst. 29*, 2 (May 2011), 6:1–6:33.

[15] Druschel, P., and Peterson, L. L. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *In Proceedings of the Fourteenth ACM symposium on Operating Systems Principles* (1993), pp. 189–202.

[16] DUNKELS, A. Full TCP/IP for 8-bit architectures. In *International Conference on Mobile Systems, Applications, and Services* (2003).

[17] GETTYS, J., AND NICHOLS, K. Bufferbloat: Dark Buffers in the Internet. *Queue 9*, 11 (Nov. 2011), 40:40–40:54.

[18] GIACOMONI, J., MOSELEY, T., AND VACHHARAJANI, M. Fast-Forward for Efficient Pipeline Parallelism: A Cache-optimized Concurrent Lock-free Queue. In *PPoPP* (2008).

[19] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Safe and Automatic Live Update for Operating Systems. In *Proceedings of ASPLOS-XVIII* (2013).

[20] HAN, S., MARSHALL, S., CHUN, B.-G., AND RATNASAMY, S. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the USENIX conf. on Oper. Sys. Design and Impl.* (2012).

[21] HRUBY, T., BOS, H., AND TANENBAUM, A. S. When Slower is Faster: On Heterogeneous Multicores for Reliable Systems. In *Proceedings of USENIX ATC* (San Jose, CA, 2013), pp. 255–266.

[22] HRUBY, T., VOGT, D., BOS, H., AND TANENBAUM, A. S. Keep Net Working - On a Dependable and Fast Networking Stack. In *Proceedings of Dependable Systems and Networks (DSN 2012)* (Boston, MA, June 2012).

[23] JEONG, E., WOOD, S., JAMSHED, M., JEONG, H., IHM, S., HAN, D., AND PARK, K. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014).

[24] LEMON, J. Kqueue - A Generic and Scalable Event Notification Facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (2001).

[25] MARINOS, I., WATSON, R. N. M., AND HANDLEY, M. Network Stack Specialization for Performance. In *Proceedings of the Workshop on Hot Topics in Networks* (2013).

[26] NIKOLAEV, R., AND BACK, G. VirtuOS: An Operating System with Kernel Virtualization. In *Proceedings of SOSP* (2013).

[27] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013).

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

[29] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Trans. Comput. Syst. 18*, 1 (2000).

[30] RIZZO, L. Netmap: A Novel Framework for Fast Packet I/O. In *Proc. of the 2012 USENIX conference on Annual Technical Conference* (2012).

[31] RIZZO, L., LETTIERI, G., AND MAFFIONE, V. Speeding Up Packet I/O in Virtual Machines. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems* (2013).

[32] SASTRY, D. C., AND DEMIRCI, M. The QNX Operating System. *Computer 28* (November 1995).

[33] SHALEV, L., SATRAN, J., BOROVIK, E., AND BEN-YEHUDA, M. IsoStack: Highly Efficient Network Processing on Dedicated Cores. In *Proceedings of the USENIX Annual Technical Conference* (2010).

[34] 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).

[35] SOARES, L., AND STUMM, M. Exception-less System Calls for Event-Driven Servers. In *Proc. of the USENIX Annual Technical Conf.* (2011).