On Sockets and System Calls

Minimizing Context Switches for Sockets

Tomáš Hrubý Teodor Crivăţ Herbert Bos
Andrew S. Tanenbaum

Vrije Universiteit Amsterdam

POSIX Socket API

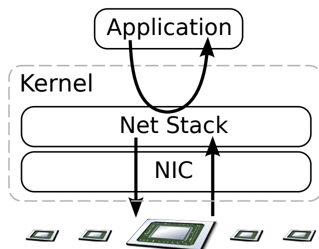
POSIX sockets are de facto standard for network applications

- Well understood API
- Used by legacy software
- Portable across operating systems

POSIX Socket API

The Sockets are also problematic!

- Each socket API call is a system call
- System calls are disruptive!
- OS execution interleaves with applications
- Nonblocking socket calls are expensive



POSIX Socket API

The widely accepted solution is inventing new APIs

- Non-POSIX interfaces
- Breaks portability!
- Legacy code does not benefit

Socket API - current solutions

- Relaxing of file-based POSIX Socket API
- Batching of system calls
- Per-core channels to the OS (kernel)
- Polling - timers & thresholds

Our hypothesis :

The problem is just the implementation not the API itself!

System Calls Overhead

Problem is the system call!

- *Synchronous* execution transfer from applications to the OS
- CPU switches stacks and instruction pointers
- OS saves and restores the application context
- CPU structures are thrashed by switching execution streams
- Recovering optimal performance takes time

Socket API - event-driven servers

- High-performance apps use event-driven model
- Single thread per core
- Apps require efficient way of testing readiness of sockets

```
1   int main(int argc, char ** argv)
2   {
3       ...
4       for(;;) {
5           socket s;
6
7           poll_all_sockets(); /* block until ready */
8
9           for_each_ready_socket(&s)
10              do_action(&s);
11      }
12      ...
13  }
```

Socket API - event-driven servers

```
1  do_action(socket * s)
2  {
3      ...
4      } else if (socket_is_listening(s)) {
5          /* accept at most 'limit' connections */
6          for (i = 0; i < limit; i++) {
7              accept(s); /* try to accept a connection ... */
8              if (errno == EAGAIN)
9                  return; /* ... and stop if it would block */
10             try_read_data(s); /* non-blocking read test */
11         }
12     }
13     ...
14 }
```

- Applications often try if a call would succeed
- Cycles wasted when many speculative calls fail
- Latency increases

The challenge :

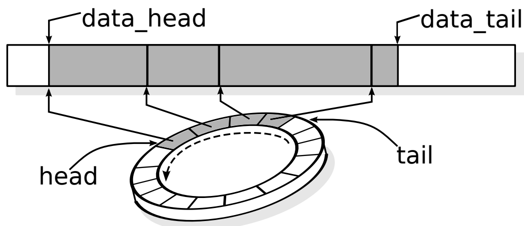
Eliminate system calls from sockets!

Exposed socket buffers

What if applications could see what is in the buffers w/o asking?

- Most calls are resolved in user space
- Failing nonblocking calls is almost free!
- Reads and writes straight from/to the socket buffers
- System calls used only to block when there is no work!

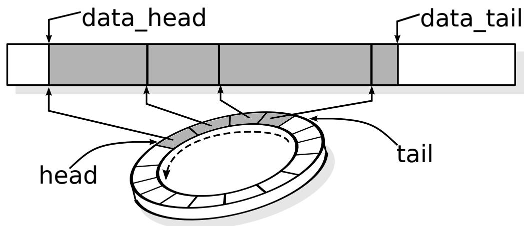
Exposed socket buffers



Each socket is :

- memory shared between OS and the application
- 2 unidirectional lock-less queues / rings for descriptors
- 2 memory areas for allocating data

Exposed socket buffers



The application can simply test :

- if the queue is empty when reading / accepting
- if the queue is full when writing
- How much space or data is in the buffer

Exposed socket buffers

```
1    /* accept at most 'limit' connections */
2    for (i = 0; i < limit; i++) {
3        accept(s); /* try to accept a connection ... */
4        if (errno == EAGAIN)
5            return; /* ... and stop if it would block */
6    }
```

Testing the socket again is cheap!

Signaling - how does it work?

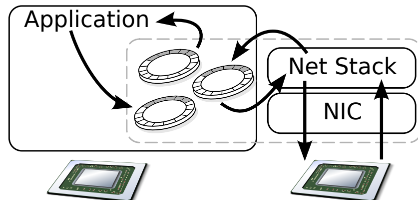
- Apps poll as long as there is a ready socket
- Apps block only if scanning sockets in user space fails
- Apps block on `select`-like system calls (or `epoll`, `kqueue, ...`)
- Blocking calls internally use the “`select`”
- System wakes up the apps when sockets ready again

The network stack cannot poll easily without disturbing the apps!

Network stack on a different core

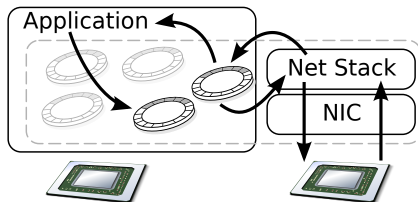
Let's place the stack on a dedicated core!

- The network stack can poll
- No interleaving with applications
- Only if an application is active, check the sockets
- When the network stack blocks, it uses `MWAIT`
- Applications can notify the network stack by a memory write



Mapping exposed socket buffers

- Mapped by the OS before apps get a handle
- Creating new mappings is slow
- We recycle existing mappings after closing sockets
- Reusing mappings is fast
- When many unused mappings, we unmap many at once



New Sockets - summary

We do not need system calls for :

- Reading and writing
- Polling
- Signaling to the OS

In addition :

- Copying within the address space of the applications
- OS used only for blocking

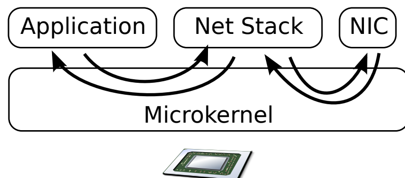
Does this actually work?

NewtOS - a multiserver system

- Multiserver system based on MINIX 3
- Implemented by isolated processes on top of a microkernel
- Multiserver systems primarily focus on reliability
- Low performance - is it inherent?

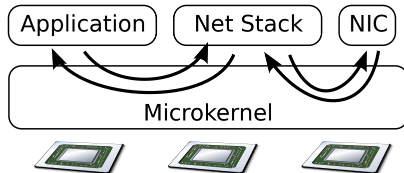
NewtOS - a multiserver system

- The network stack is in user space!
- System calls implemented by message passing
- Possibly a cascade of messages for each system call
- Execution of OS processes interleaves with apps.

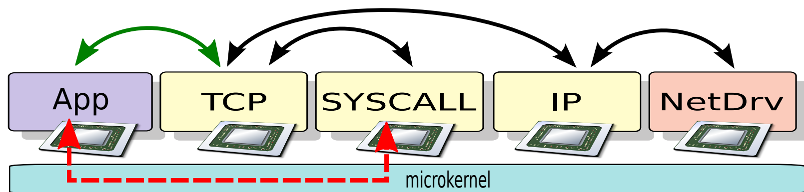


NewtOS - a multiserver system

- Multiserver systems easily embrace multicores
- Simple to run OS parts on dedicated cores



NewtOS network stack



- Network stack partitioned for improved reliability
- Individual components on dedicated cores (+ UDP and PF)
- Each system call includes SYSCALL and at least 1 server

Multiserver systems - the cost of system calls

Cycles to complete a `recvfrom()` call that returns EAGAIN

NewtOS - kernel msgs	79800
NewtOS - user space msgs	19950

NewtOS - system call involves 3 processes on 3 different cores!

Multiserver systems - the cost of system calls

Cycles to complete a `recvfrom()` call that returns EAGAIN

NewtOS - kernel msgs	79800
NewtOS - user space msgs	19950
Linux	478

NewtOS - system call involves 3 processes on 3 different cores!

Multiserver systems - the cost of system calls

Cycles to complete a `recvfrom()` call that returns EAGAIN

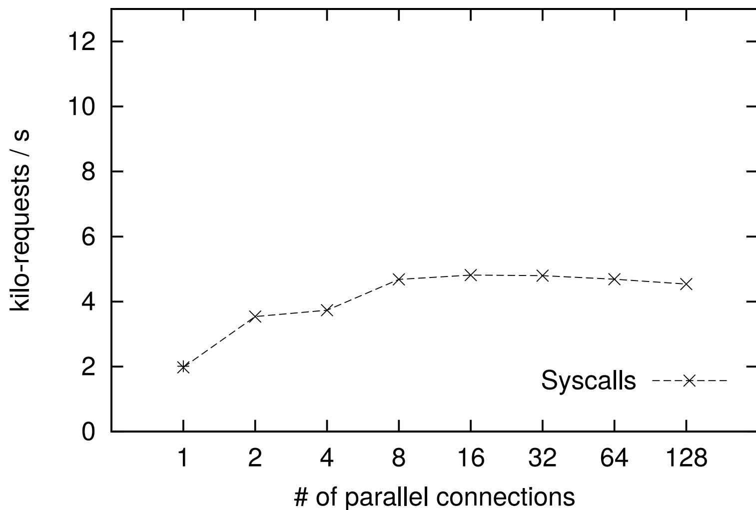
NewtOS - kernel msgs	79800
NewtOS - user space msgs	19950
Linux	478
NewtOS - no system calls	137

NewtOS - system call involves 3 processes on 3 different cores!

Evaluation - testbed

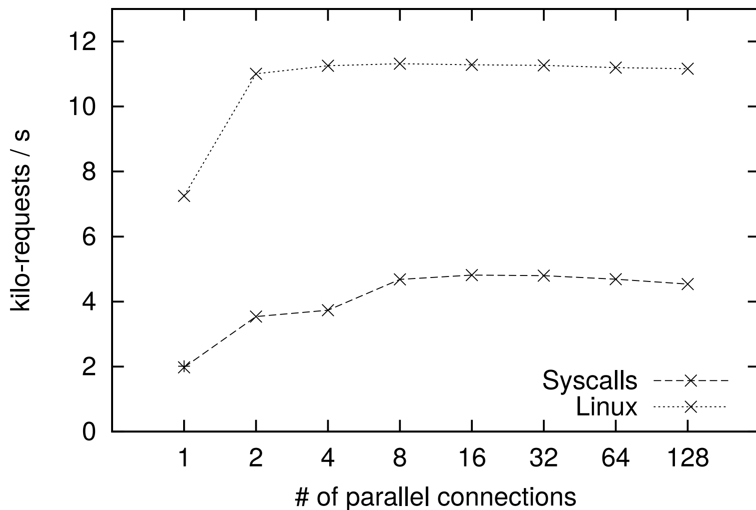
- 12-core AMD 1.9 GHz
- 10G Intel NIC
- lighttpd web server - single process
- Serving static files cached in memory

Evaluation



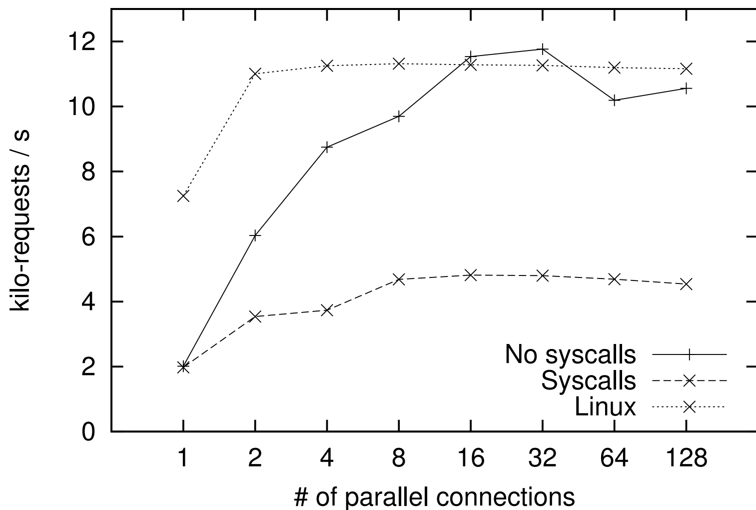
1 request per connection, 20 byte files

Evaluation



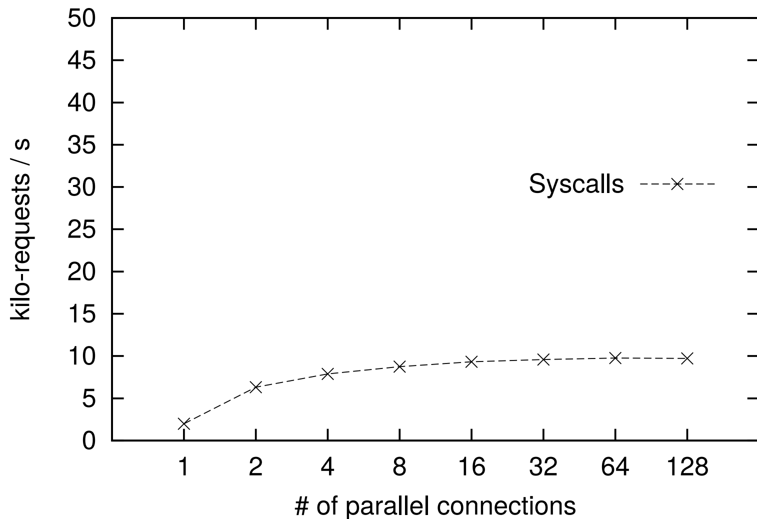
1 request per connection, 20 byte files

Evaluation



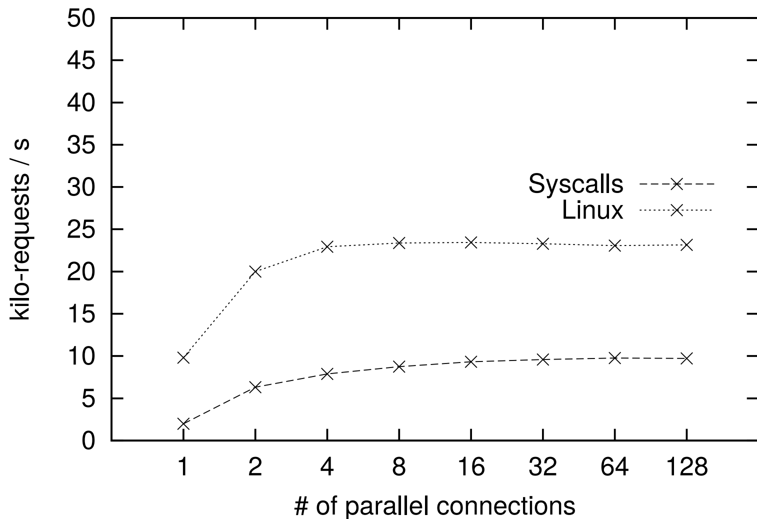
1 request per connection, 20 byte files

Evaluation



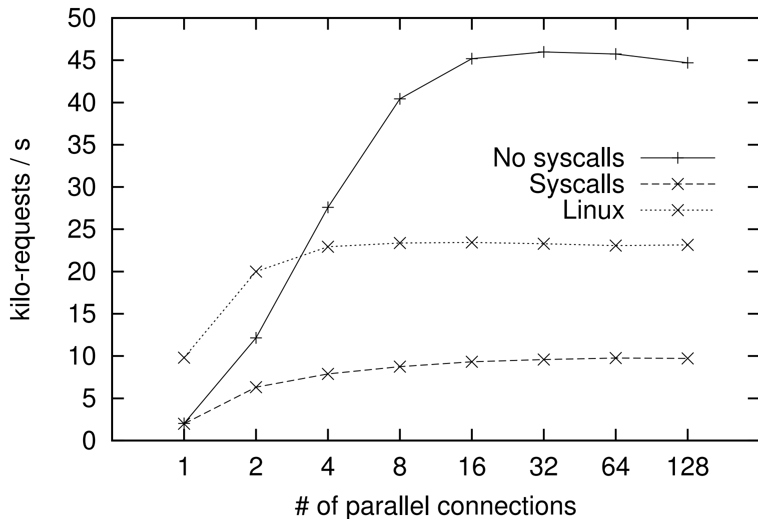
10 request per connection, 20 byte files

Evaluation



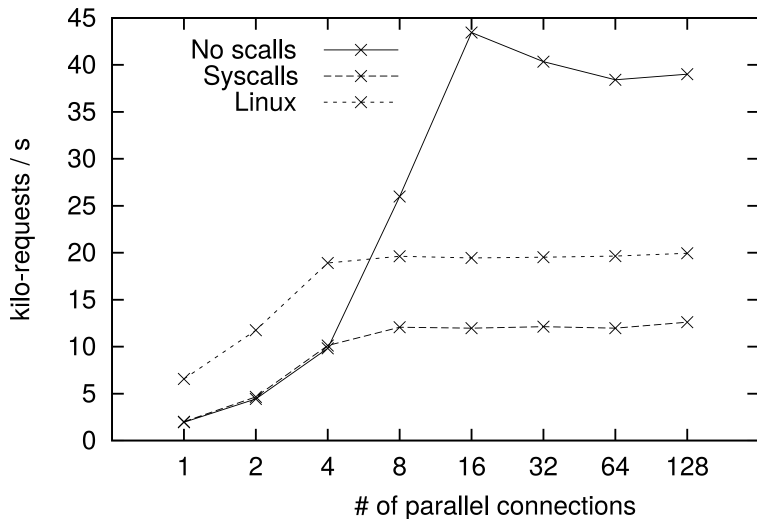
10 request per connection, 20 byte files

Evaluation



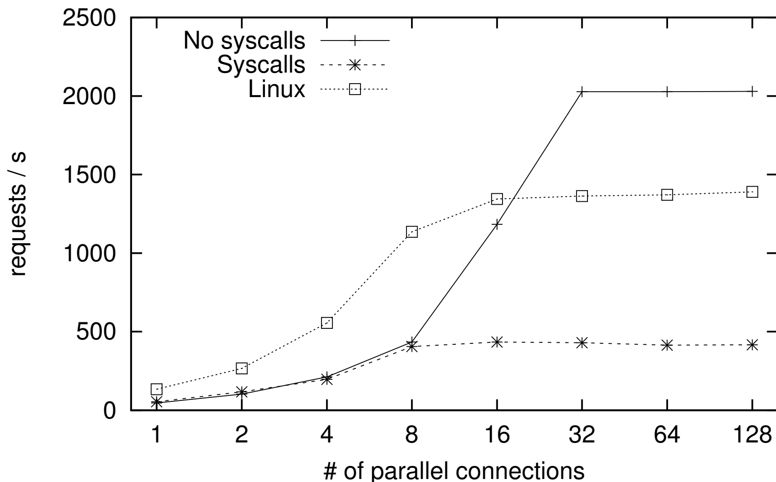
10 request per connection, 20 byte files

Evaluation



10 request per connection, 11 kB files

Evaluation



1 request per connection, 512 kB files

Benefits of running system and applications on different cores

- Applications and the system can use entire cores
- Smaller instruction cache miss rate

Linux	8.5%
NewtOS - system calls	4.5%
NewtOS - w/o system calls	1.5%

`lighttpd` instruction cache miss rate

Limitations

We cannot handle `fork`

- Shared memory would be used by independent processes
- We revert to legacy sockets - OS is responsible for synchronization
- `fork` is expensive, outweighs benefits for short-lived connections
- We can use `ioctl` as a workaround - requires code changes in legacy applications

Related work

FlexSC - OSDI'10

- General mechanism decouples invocation from execution
- Batching as many syscalls as possible in shared memory

MegaPipe - OSDI'12

- Relaxes file-based POSIX Socket API - different API !!!

IsoStack - ATC'10

- Network stack polls on a dedicated core
- No contention on data structures accessed by many cores

Conclusions

Network stack on a different core and exposed socket buffers make the following possible :

- We can avoid 99% of system calls when load is high
- Applications can read information directly from the sockets
- Speculative nonblocking API calls are cheap
- Execution of applications and OS does not interleave

The new socket design makes networking in reliable multiserver systems competitive with commodity operating systems!