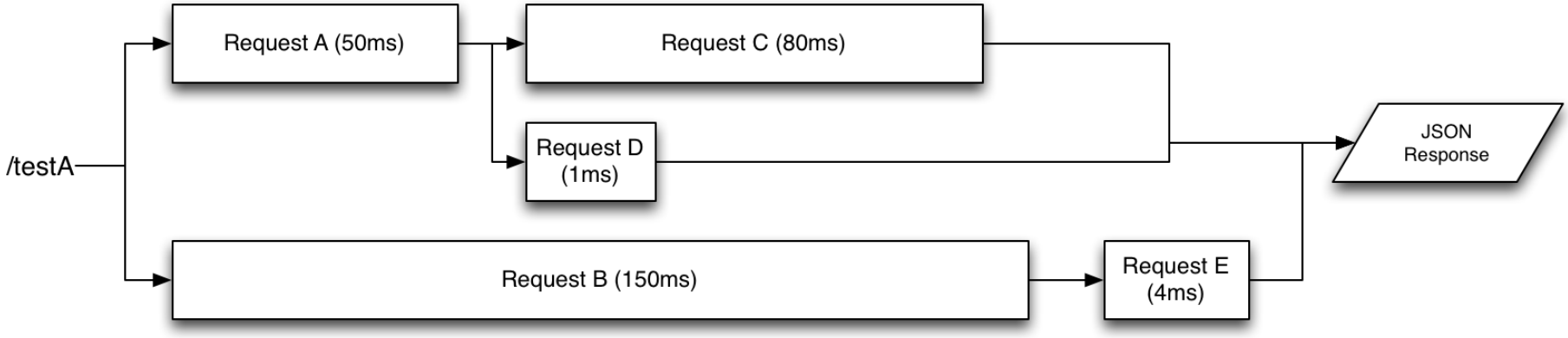
RxNetty vs Tomcat
Performance Results

Brendan Gregg; Performance and Reliability Engineering
Nitesh Kant, Ben Christensen; Edge Engineering
updated: Apr 2015

Results based on

- The “Hello Netflix” benchmark (wsperflab)
- Tomcat
- RxNetty
- physical PC
  - Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz: 4 cores, 1 thread per core
- OpenJDK 8
  - with frame pointer patch
- Plus testing in other environments

Hello Netflix



RxNetty vs Tomcat performance

In a variety of tests, RxNetty has been faster than Tomcat. This study covers:

1. What specifically is faster?
2. By how much?
3. Why?

1. What specifically is faster?

1. What specifically is faster?

- CPU consumption per request
  - RxNetty consumes less CPU than Tomcat
  - This also means that a given server (with fixed CPU capacity) can deliver a higher maximum rate of requests per second
- Latency under load
  - Under high load, RxNetty has a lower latency distribution than Tomcat

2. By how much?

2. By how much?

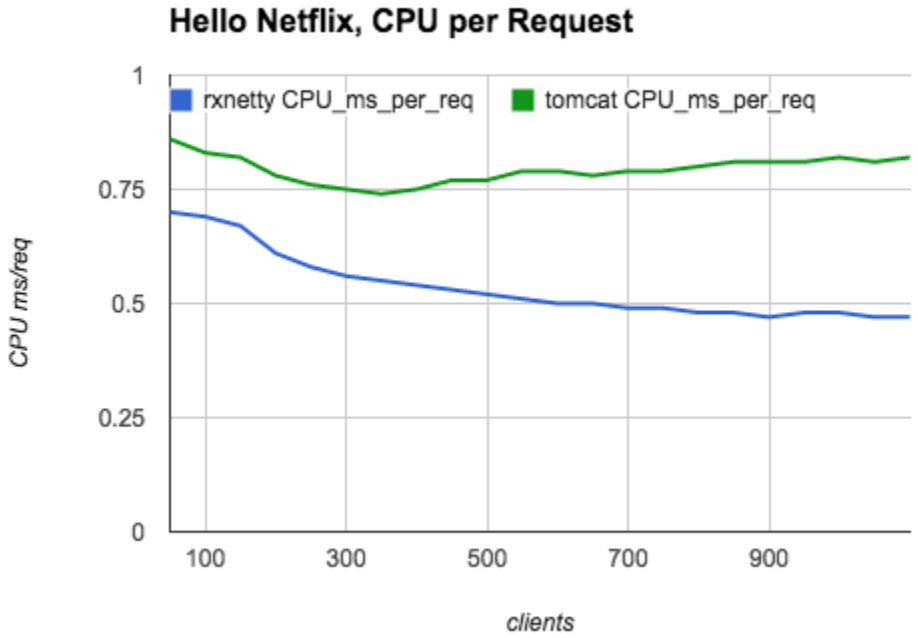
The following 5 graphs show performance vs load (clients)

1. CPU consumption per request
2. CPU resource usage vs load
3. Request rate
4. Request average latency
5. Request maximum latency

Bear in mind these results are for this environment, and this workload

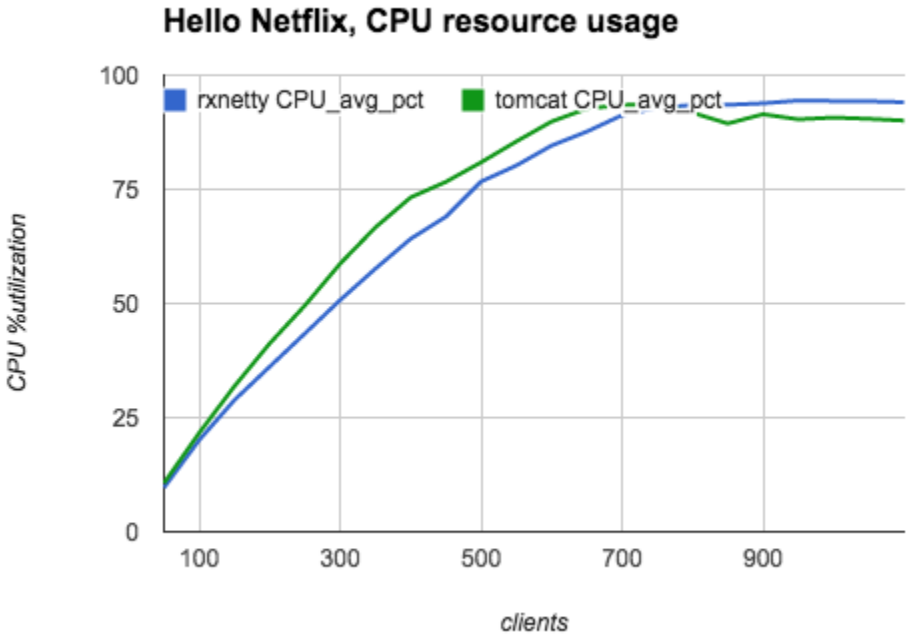
2.1. CPU Consumption Per Request

- RxNetty has generally lower CPU consumption per request (over 40% lower)
- RxNetty keeps getting faster under load, whereas Tomcat keeps getting slower



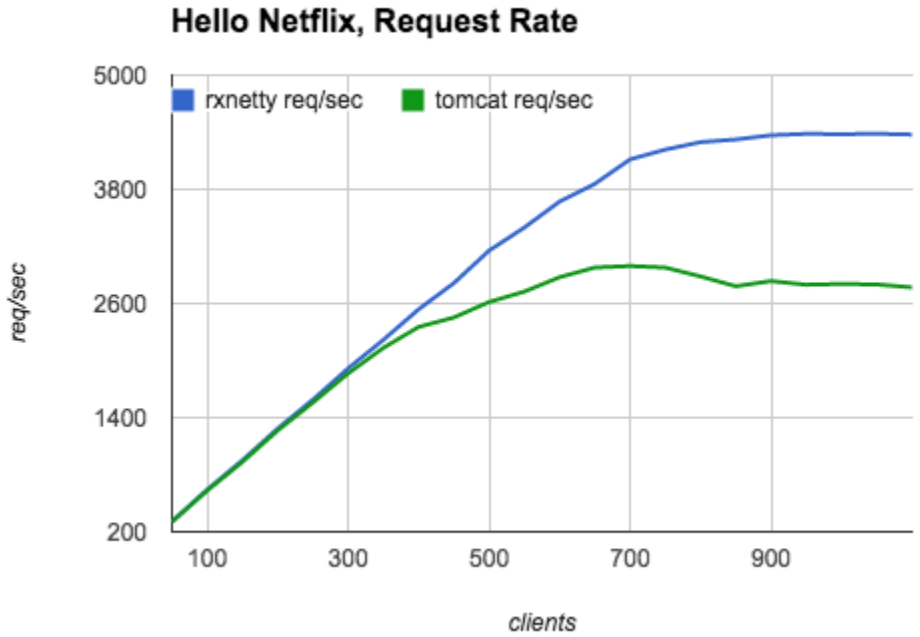
2.2. CPU Resource Usage vs Load

- Load testing drove the server's CPUs to near 100% for both frameworks



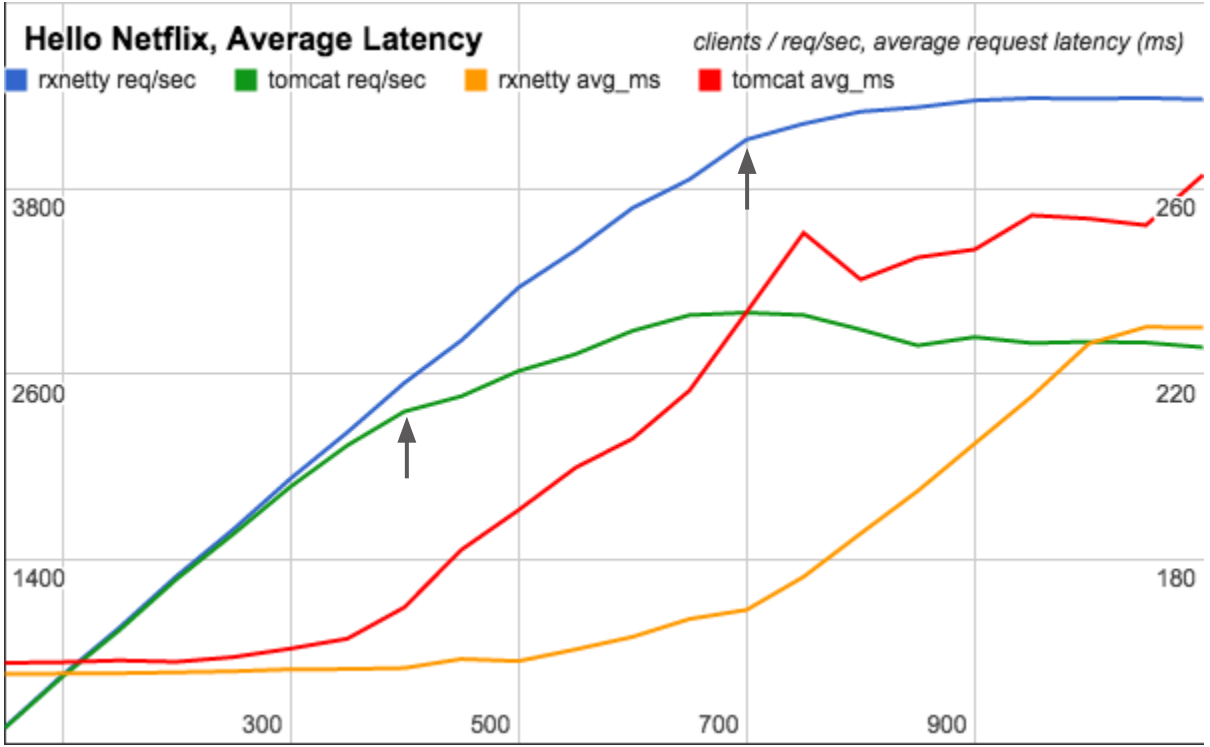
2.3. Request Rate

- RxNetty achieved a 46% higher request rate
- This is mostly due to the lower CPU consumption per request



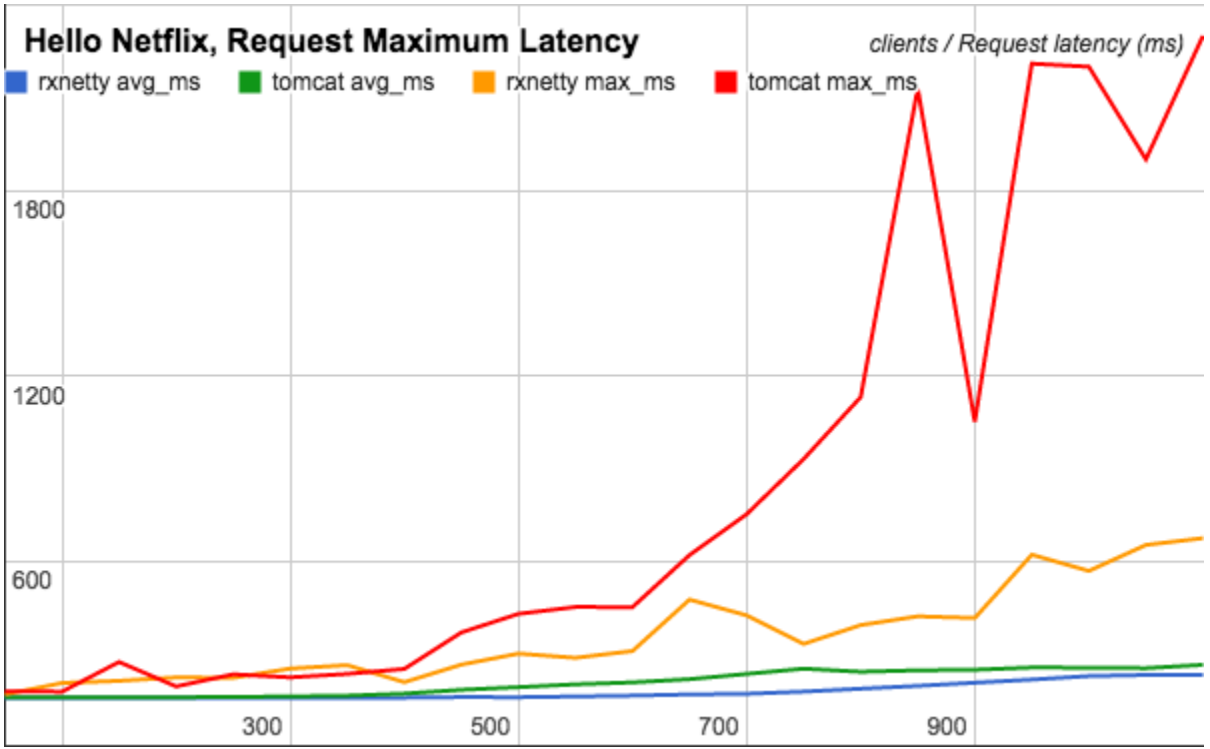
2.4. Request Average Latency

- Average latency increases past the req/sec knee point (when CPU begins to be saturated)
- RxNetty's latency breakdown happens with much higher load



2.5. Request Maximum Latency

- The degradation in maximum latency for Tomcat is much more severe



3. Why?

3. Why?

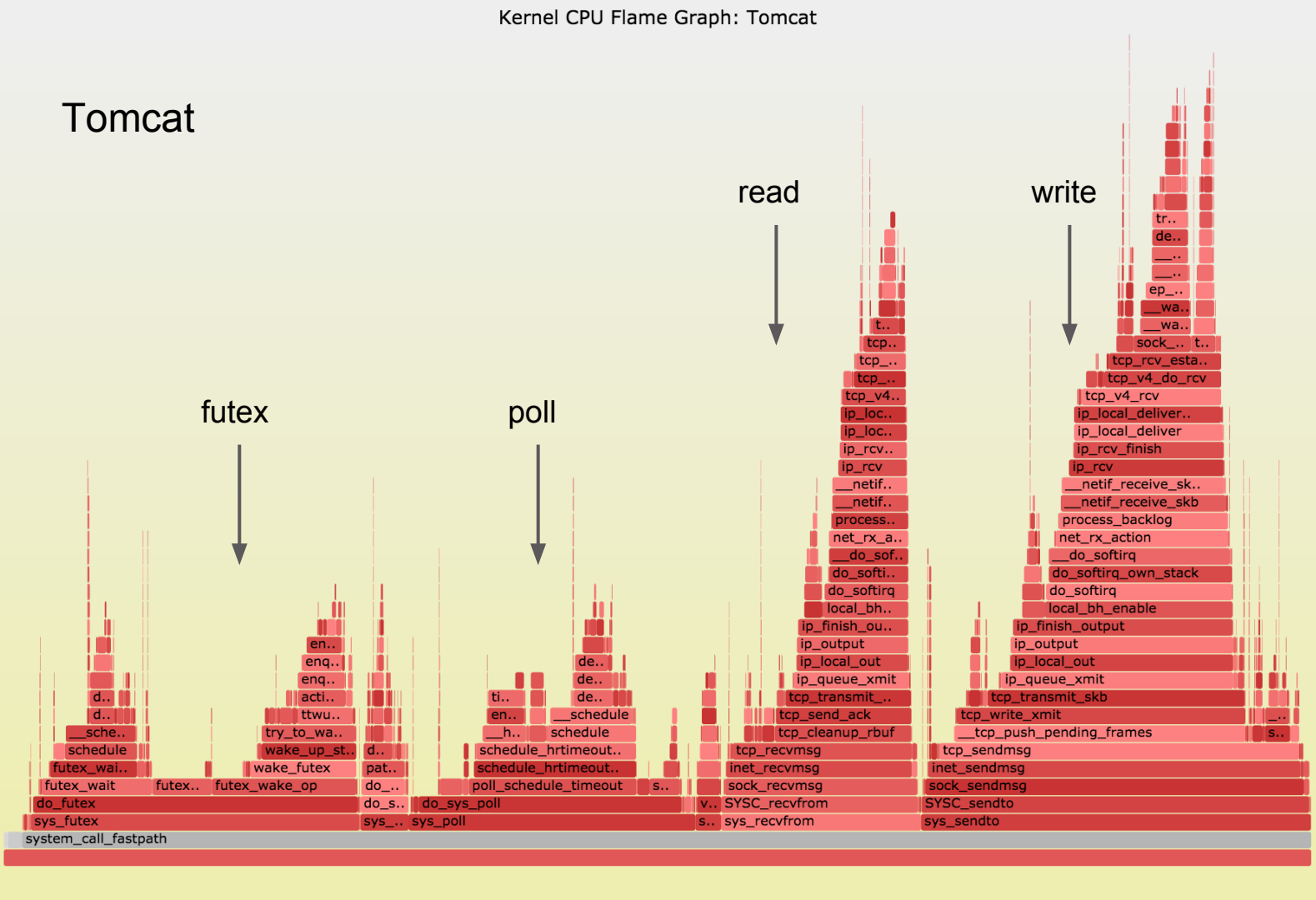
1. CPU consumption per request
    - RxNetty is lower due to its framework code and lower object allocation rate, which in turn reduces GC overheads
    - RxNetty also trends lower due to its event loop architecture, which reduces thread migrations under load, which improves CPU cache warmth and memory locality, which improves CPU Instructions Per Cycle (IPC), which lowers CPU cycle consumption per request
2. Lower latencies under load
    - Tomcat has higher latencies under load due to its thread pool architecture, which involves thread pool locks (and lock contention) and thread migrations to service load

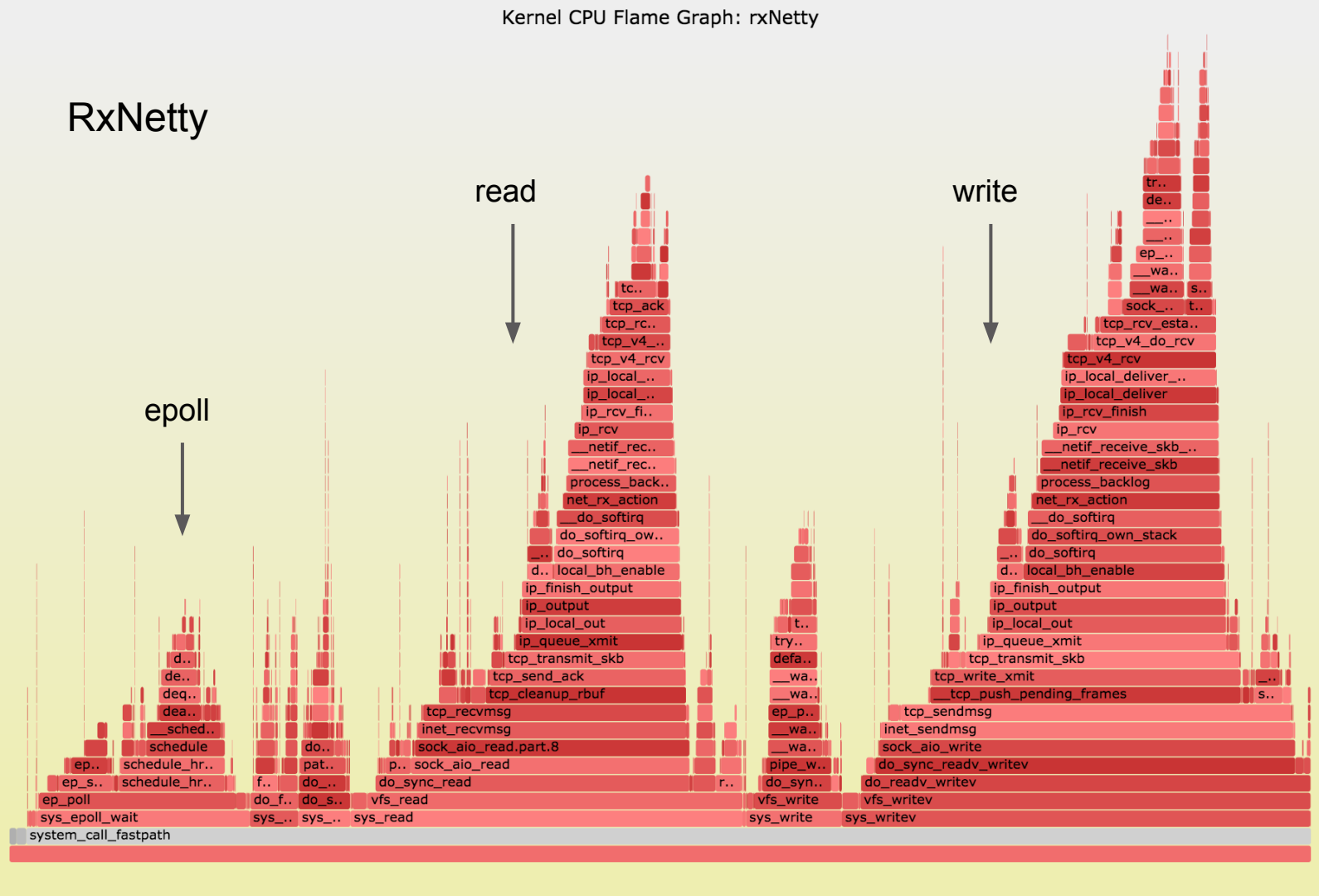
3.1. CPU Consumption Per Request

Studied using:

1. Kernel CPU flame graphs
2. User CPU flame graphs
3. Migration rates
4. Last Level Cache (LLC) Loads & IPC
5. IPC & CPU per request

3.1.1. Kernel CPU Flame Graphs


Kernel CPU Flame Graph: Tomcat
Tomcat
futex
poll
read
write
system_call_fastpath
sys_futex
do_futex
futex_wait
futex_wai..
schedule
__sche..
d..
d..
futex..
futex_wake_op
wake_futex
wake_up_st..
try_to_wa..
ttwu..
acti..
enq..
enq..
en..
do_..
do_s..
pat..
d..
sys_..
sys_poll
do_sys_poll
poll_schedule_timeout
schedule_hrtimeout..
schedule_hrtimeout..
__h..
schedule
__schedule
en..
ti..
de..
de..
de..
s..
s..
v..
sock_recvmsg
inet_recvmsg
tcp_recvmsg
tcp_cleanup_rbuf
tcp_send_ack
tcp_transmit_..
ip_queue_xmit
ip_local_out
ip_output
ip_finish_ou..
local_bh..
do_softirq
do_softi..
__do_sof..
net_rx_a..
process..
__netif..
__netif..
ip_rcv
ip_rcv..
ip_loc..
ip_loc..
tcp_v4..
tcp_..
tcp_..
tcp..
t..
SYSC_recvfrom
sys_recvfrom
SYSC_sendto
sys_sendto
inet_sendmsg
sock_sendmsg
tcp_sendmsg
__tcp_push_pending_frames
tcp_write_xmit
tcp_transmit_skb
ip_queue_xmit
ip_local_out
ip_output
ip_finish_output
local_bh_enable
do_softirq
do_softirq_own_stack
__do_softirq
net_rx_action
process_backlog
__netif_receive_skb
__netif_receive_sk..
ip_rcv
ip_rcv_finish
ip_local_deliver
ip_local_deliver..
tcp_v4_rcv
tcp_v4_do_rcv
tcp_rcv_esta..
sock_..
t..
__wa..
__wa..
ep_..
__..
__..
de..
tr..
s..
_..


Kernel CPU Flame Graph: rxNetty
RxNetty
epoll
read
write
system_call_fastpath
sys_epoll_wait
sys_read
sys_write
sys_writev
ep_poll
vfs_read
vfs_write
vfs_writev
do_sync_read
do_sync_readv_writev
do_readv_writev
sock_aio_read.part.8
sock_aio_write
tcp_recvmsg
tcp_sendmsg
tcp_cleanup_rbuf
tcp_write_xmit
tcp_send_ack
tcp_transmit_skb
ip_queue_xmit
ip_output
ip_local_out
ip_finish_output
local_bh_enable
do_softirq
__do_softirq
net_rx_action
process_backlog
__netif_receive_skb
ip_rcv
ip_rcv_finish
ip_local_deliver
tcp_v4_rcv
tcp_v4_do_rcv
tcp_rcv_esta..
tcp_ack
schedule
__sched..
schedule_hr..
ep_s..
ep..
deq..
dea..
de..
d..
pipe_w..
do_syn..
inet_recvmsg
inet_sendmsg
__tcp_push_pending_frames
ip_local_deliver_..
__netif_receive_skb_..
do_softirq_own_stack
ip_local_bh_enable
__netif_rec..
process_back..
do_softirq_ow..
ip_rcv_fi..
ip_local_..
tcp_v4_..
tcp_rc..
tc..
__wa..
sock_..
ep_..
tr..
de..
__..
sys_..
do_..
do..
pat..
do_s..
f..
do_f..
p..
r..
t..
try..
defa..
s..

3.1.1. Kernel CPU Time Differences

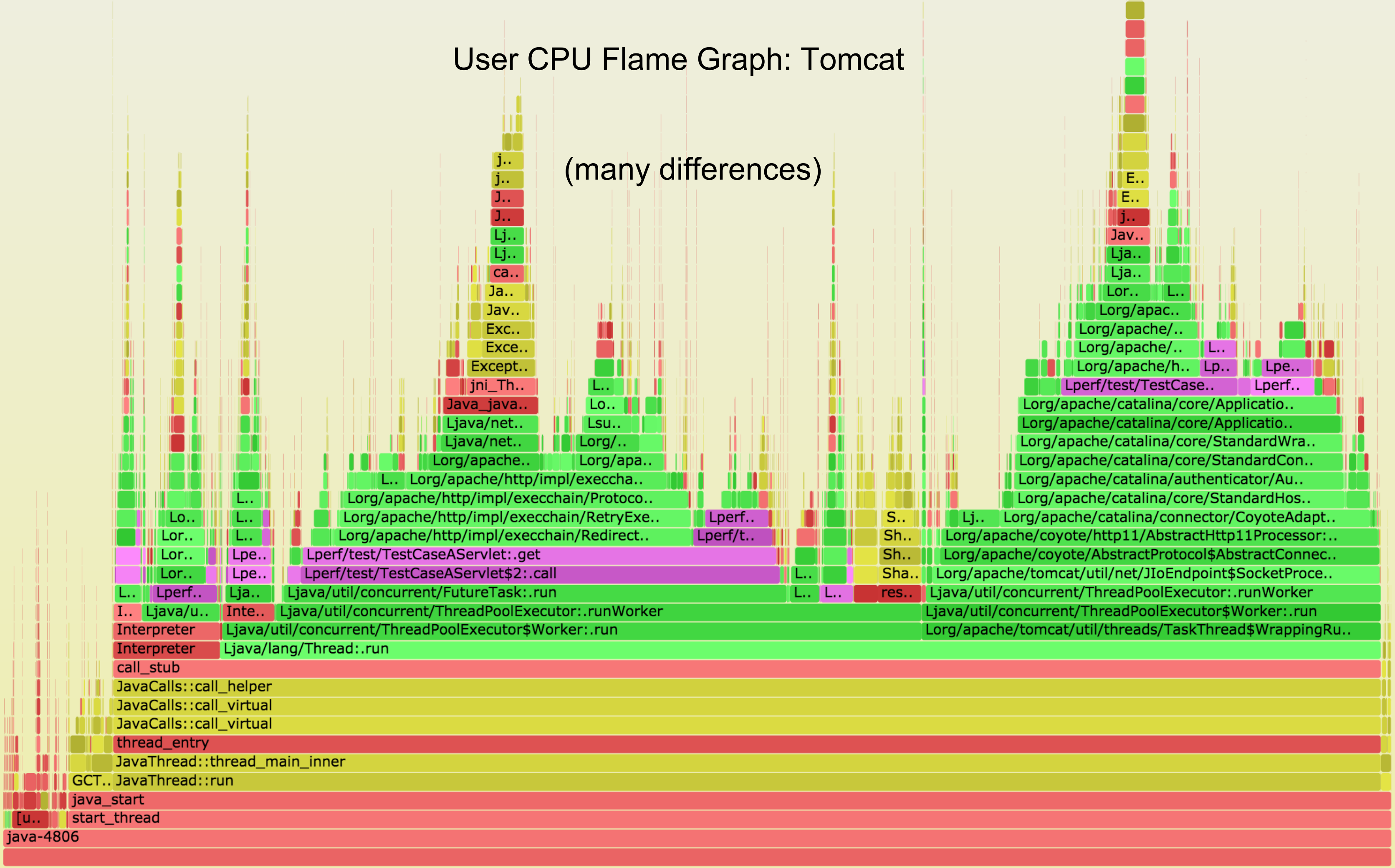
CPU system time delta per request: 0.07 ms

- Tomcat futex(), for thread pool management (0.05 ms)
- Tomcat poll() vs RxNetty epoll() (0.02 ms extra)

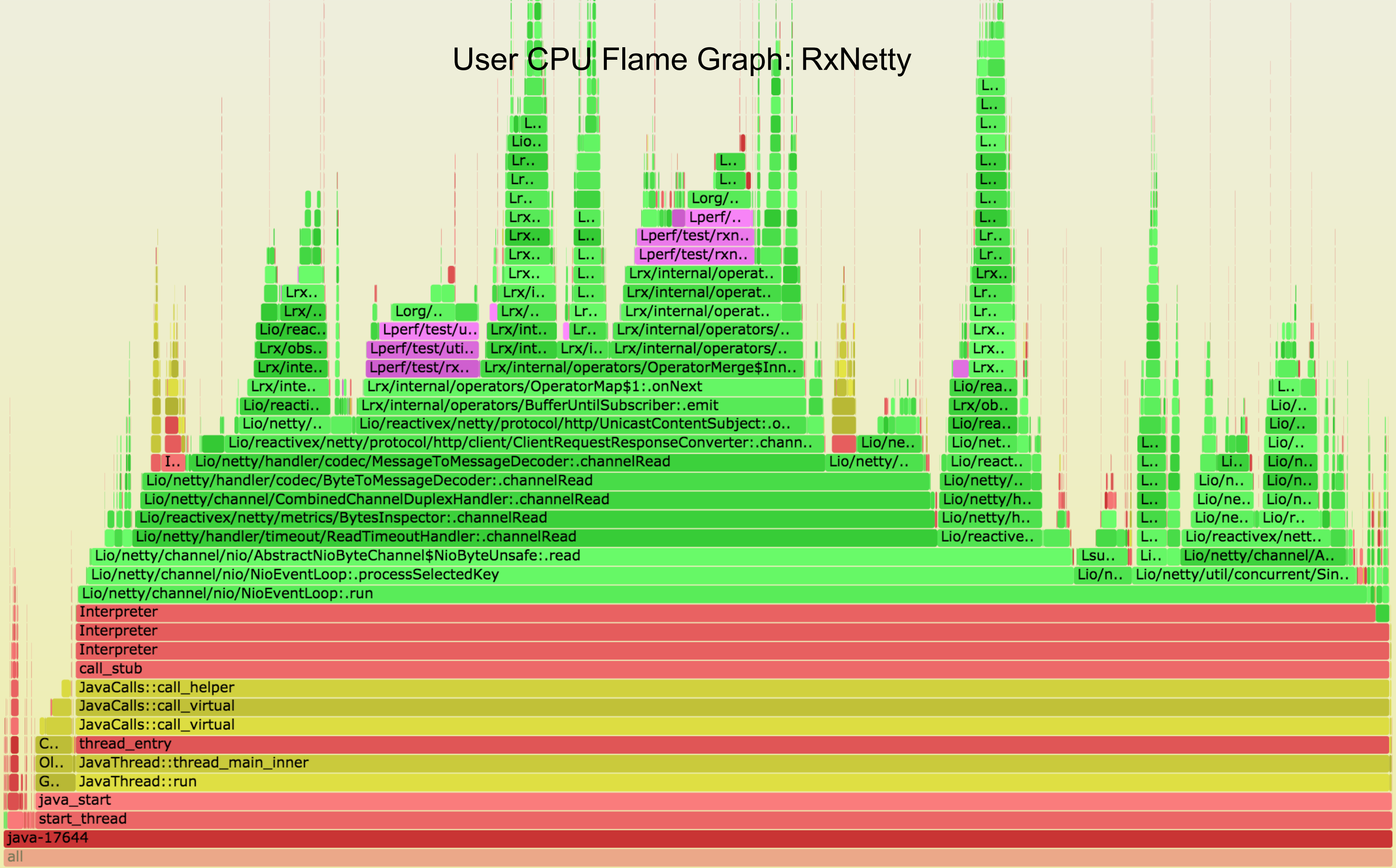
3.1.2. User CPU Flame Graphs

User CPU Flame Graph: Tomcat

(many differences)



User CPU Flame Graph: RxNetty



3.1.2. User CPU Time Differences

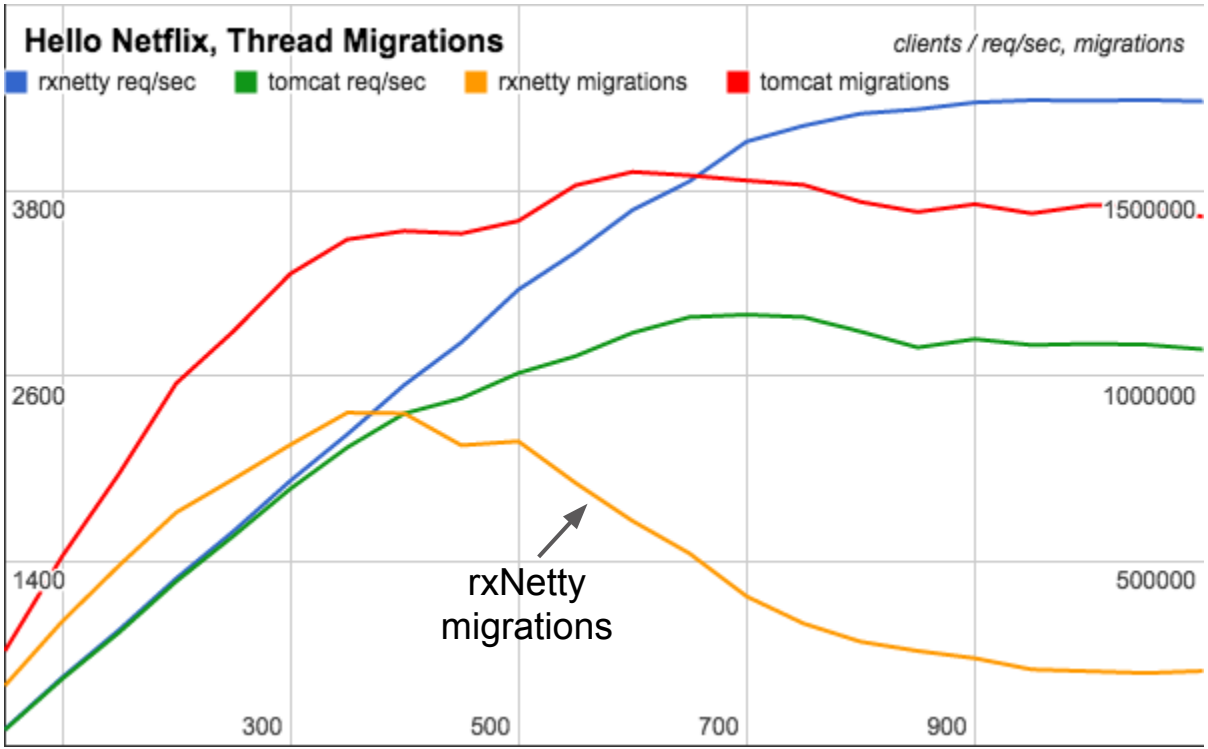
CPU user time delta per request: 0.14 ms

Differences include:

- Extra GC time in Tomcat
- Framework code differences
- Socket read library
- Tomcat thread pool calls

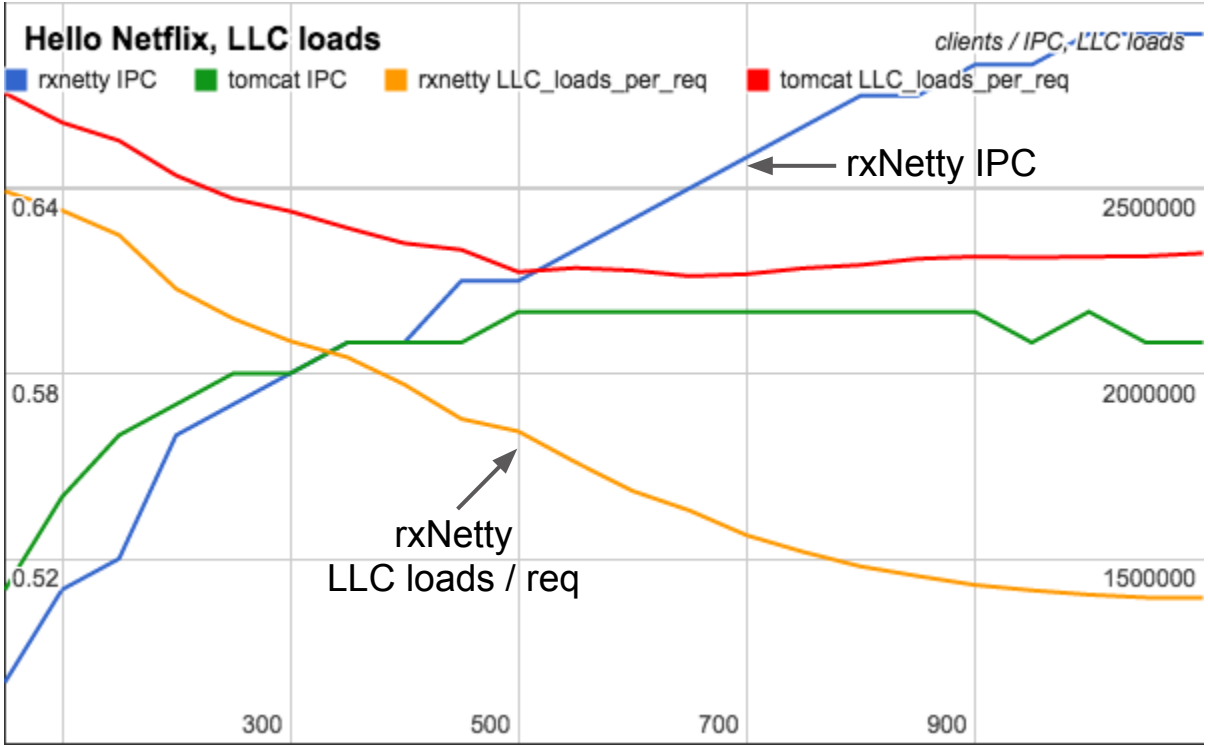
3.1.3. Thread Migrations

- As load increases, RxNetty begins to experience *lower* thread migrations
- There is enough queued work for event loop threads to keep servicing requests without switching



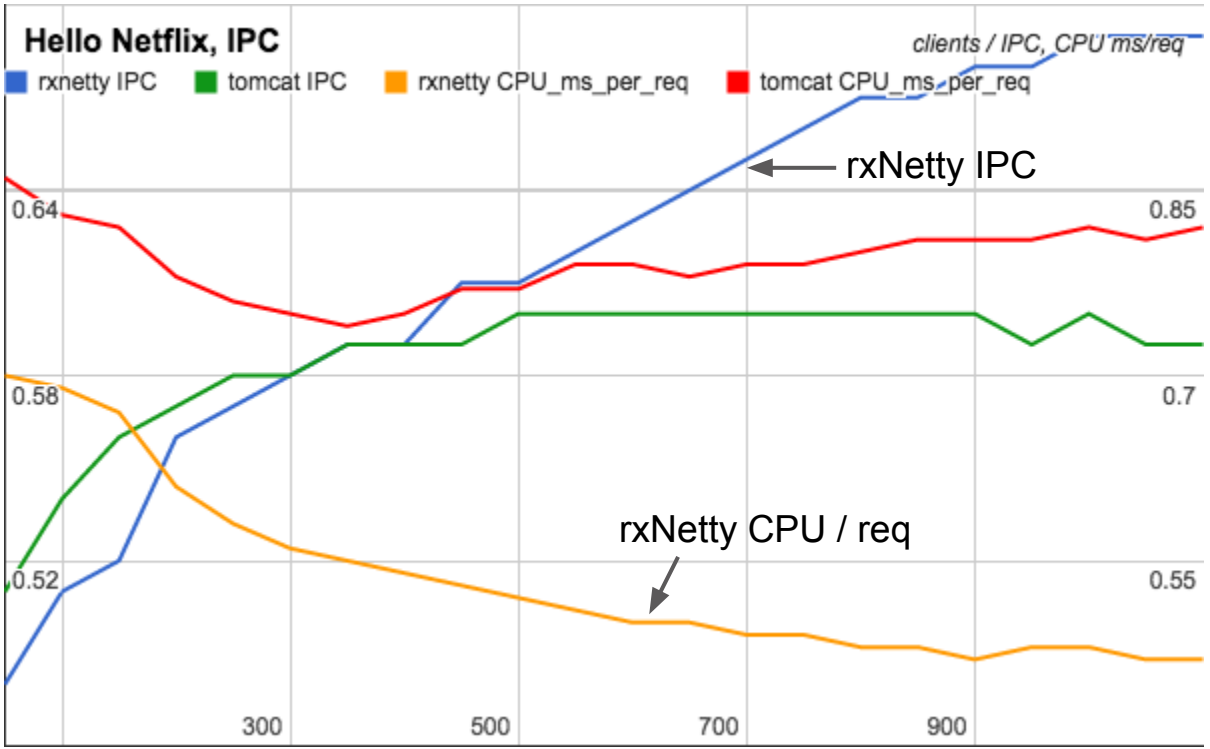
3.1.4. LLC Loads & IPC

- … The reduction in thread migrations keeps threads on-CPU, which keeps caches warm, reducing LLC loads, and improving IPC



3.1.5. IPC & CPU Per Request

- … A higher IPC leads to lower CPU usage per request



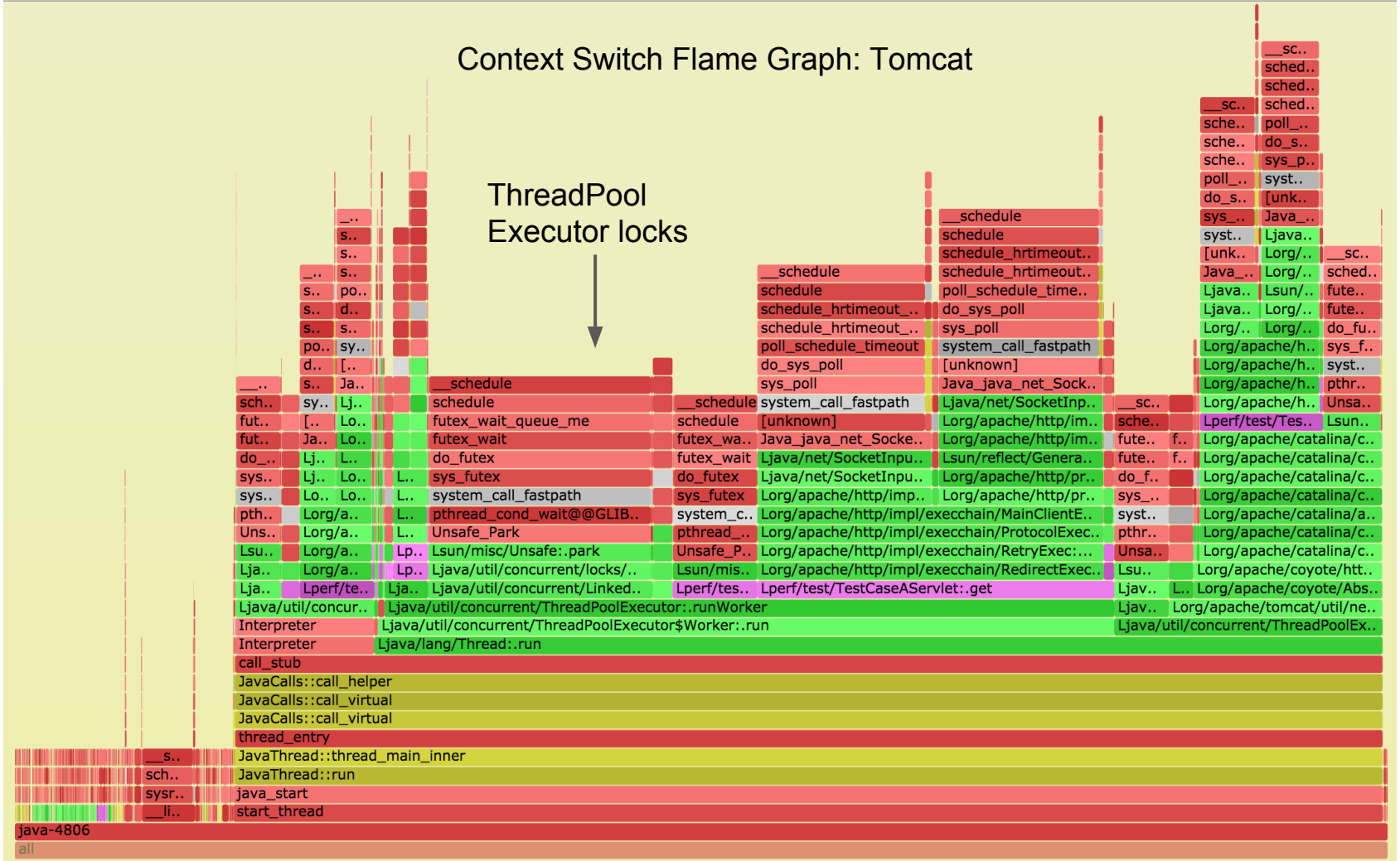
3.2. Lower Latencies Under Load

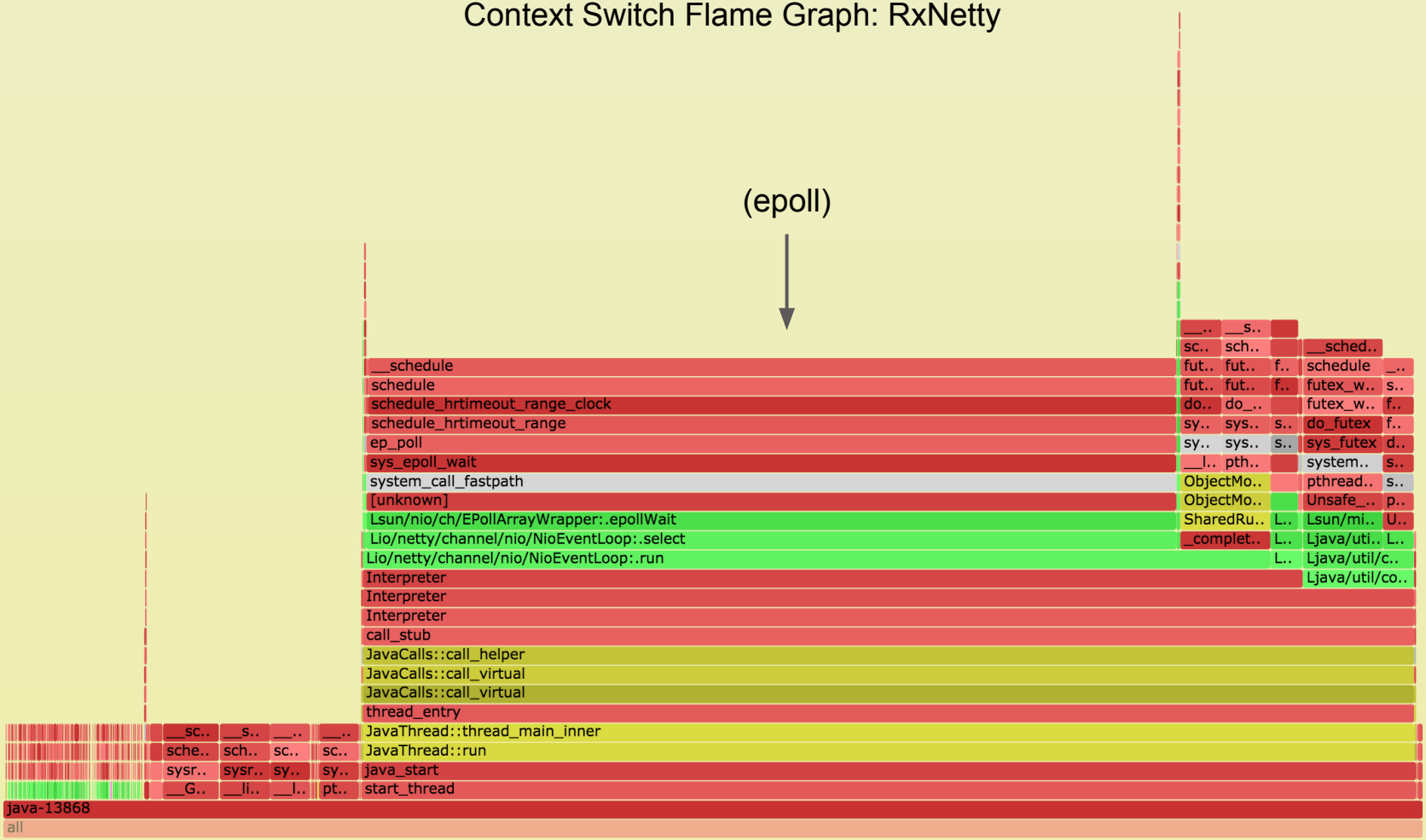
Studied using:

1. Migration rates (previous graph)
2. Context-switch flame graphs
3. Chain graphs

3.2.2. Context Switch Flame Graphs

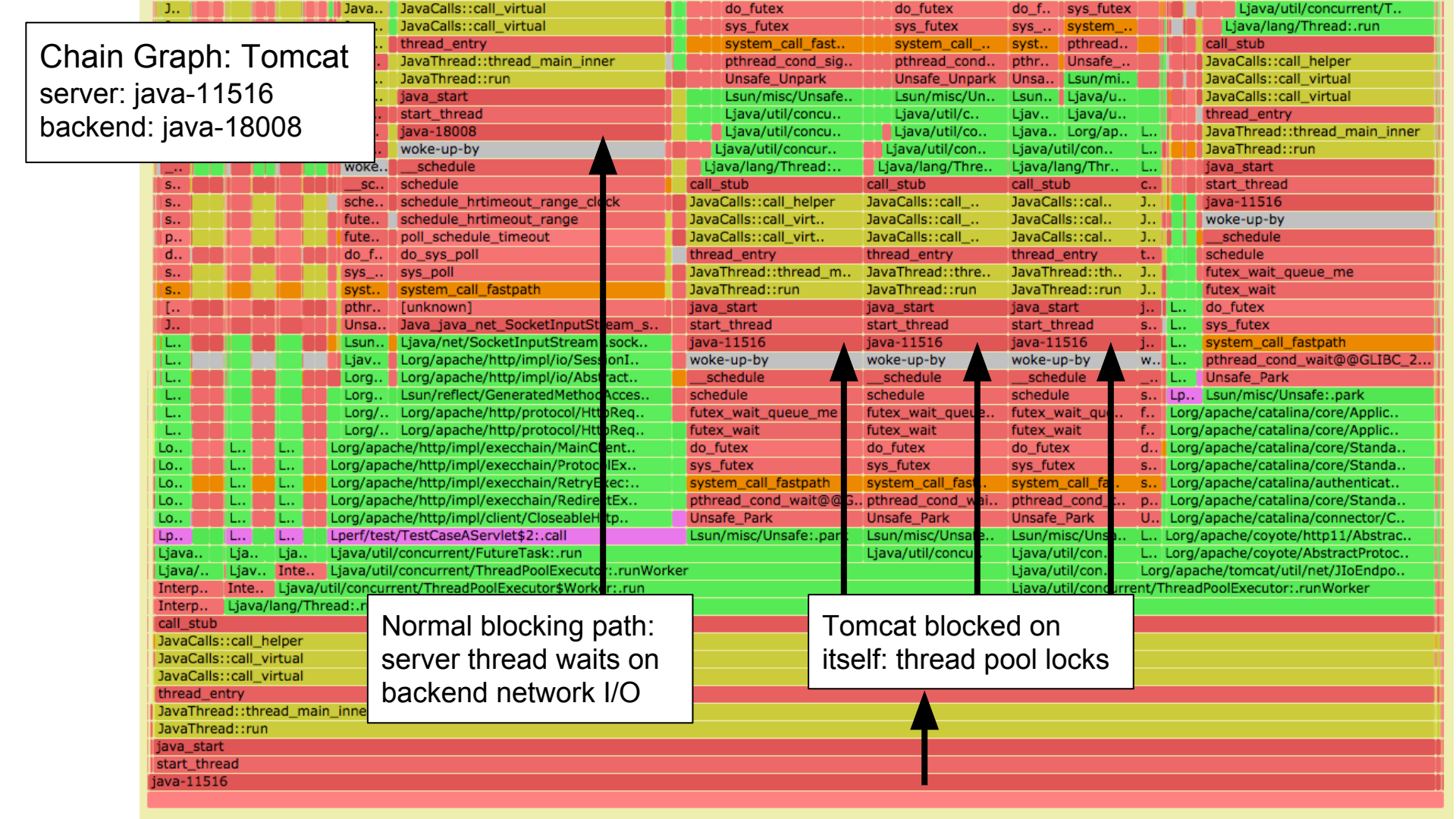
- These identify the cause of context switches, and blocking events.
  - They do not quantify the magnitude of off-CPU time; these are for identification of targets for further study
- Tomcat has additional futex context switches from thread pool management


Context Switch Flame Graph: Tomcat
ThreadPool
Executor locks
__schedule
schedule
schedule_hrtimeout..
poll_schedule_time..
do_sys_poll
sys_poll
system_call_fastpath
[unknown]
Java_java_net_Sock..
Ljava/net/SocketInp..
Lorg/apache/http/im..
Lsun/reflect/Genera..
Lorg/apache/http/pr..
futex_wait_queue_me
futex_wait
do_futex
sys_futex
pthread_cond_wait@@GLIB..
Unsafe_Park
Lsun/misc/Unsafe:.park
Ljava/util/concurrent/locks/..
Ljava/util/concurrent/Linked..
Lperf/test/TestCaseAServlet:.get
Ljava/util/concurrent/ThreadPoolExecutor:.runWorker
Ljava/util/concurrent/ThreadPoolExecutor$Worker:.run
Ljava/lang/Thread:.run
Interpreter
call_stub
JavaCalls::call_helper
JavaCalls::call_virtual
thread_entry
JavaThread::thread_main_inner
JavaThread::run
java_start
start_thread
java-4806
all


Context Switch Flame Graph: RxNetty
(epoll)
__schedule
schedule
schedule_hrtimeout_range_clock
schedule_hrtimeout_range
ep_poll
sys_epoll_wait
system_call_fastpath
[unknown]
Lsun/nio/ch/EPollArrayWrapper:.epollWait
Lio/netty/channel/nio/NioEventLoop:.select
Lio/netty/channel/nio/NioEventLoop:.run
Interpreter
Interpreter
Interpreter
call_stub
JavaCalls::call_helper
JavaCalls::call_virtual
JavaCalls::call_virtual
thread_entry
JavaThread::thread_main_inner
JavaThread::run
java_start
start_thread
java-13868
all

3.2.3. Chain Graphs

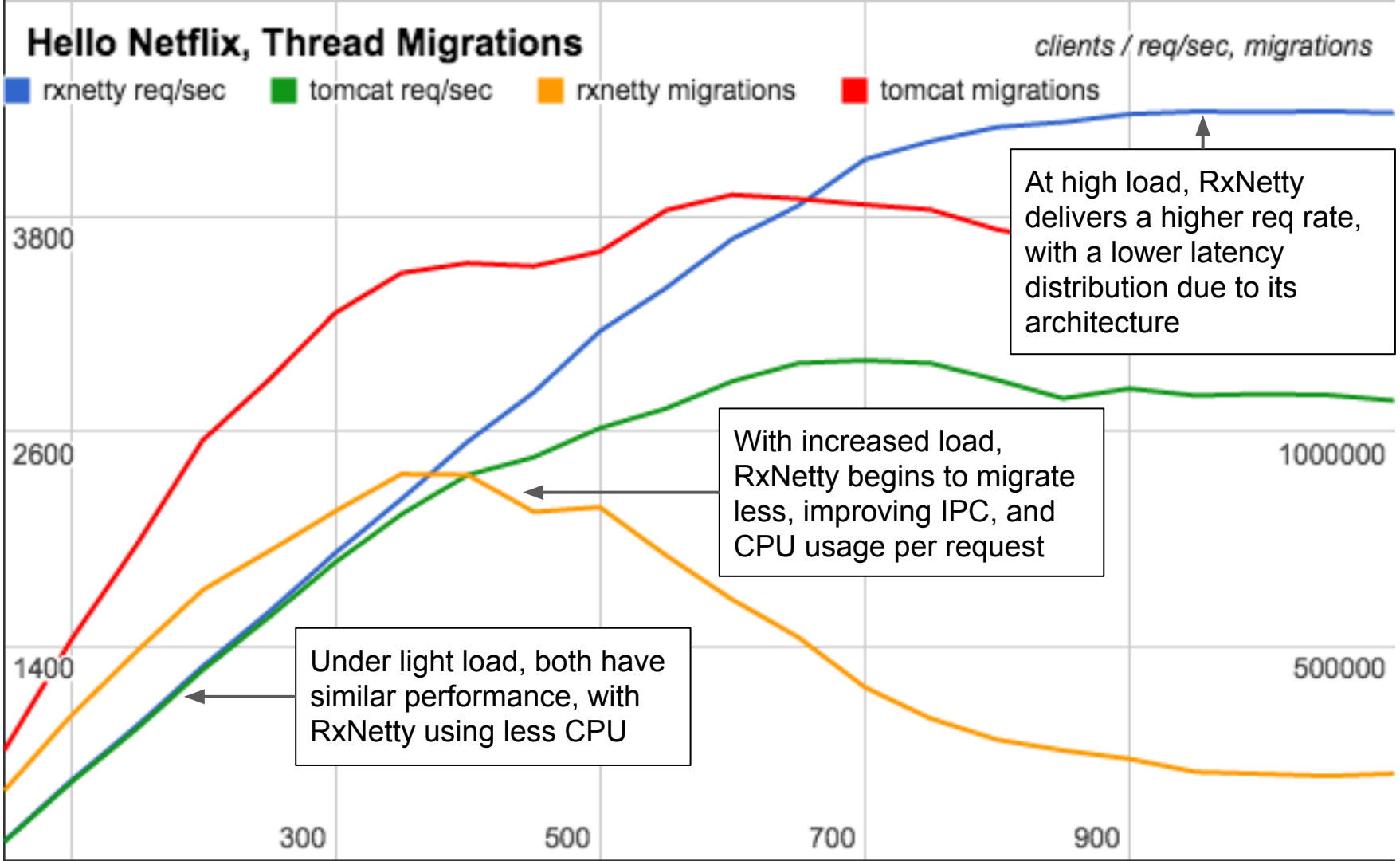
- These quantify the magnitude of off-CPU (blocking) time, and show the chain of wakeup stacks that the blocked thread was waiting on
  - x-axis: blocked time
  - y-axis: blocked stack, then wakeup stacks


Chain Graph: Tomcat
server: java-11516
backend: java-18008
Normal blocking path: server thread waits on backend network I/O
Tomcat blocked on itself: thread pool locks

Reasoning

- On a system with more CPUs (than 4), Tomcat will perform even worse, due to the earlier effects.
- For applications which consume more CPU, the benefits of an architecture change diminish.

Summary


Hello Netflix, Thread Migrations
clients / req/sec, migrations
rxnetty req/sec
tomcat req/sec
rxnetty migrations
tomcat migrations
3800
2600
1400
1000000
500000
300
500
700
900
At high load, RxNetty delivers a higher req rate, with a lower latency distribution due to its architecture
With increased load, RxNetty begins to migrate less, improving IPC, and CPU usage per request
Under light load, both have similar performance, with RxNetty using less CPU