Experiments with non-temporal writes and application architectures

Joe Damato

Hi, my name is Joe.

I work at Fastly.

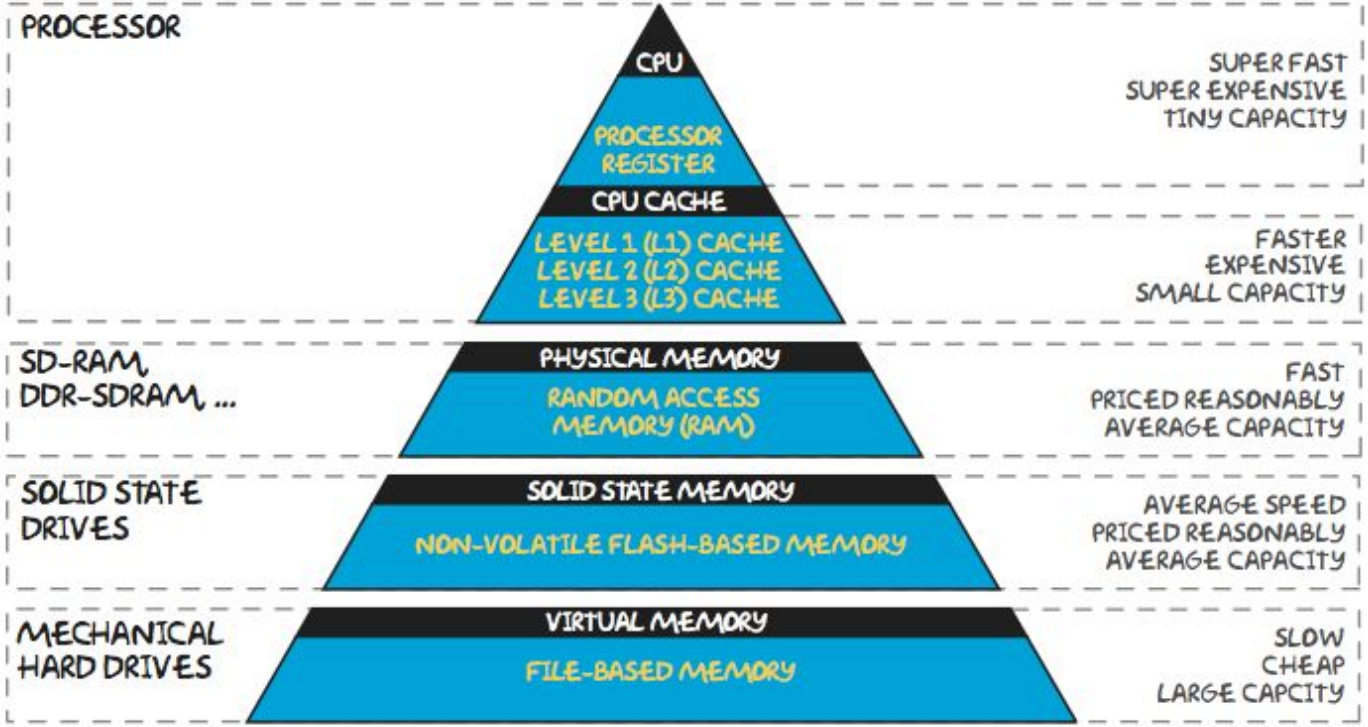
My opinions are my own.

I am *not* an x86 expert.

But I do know a little bit.

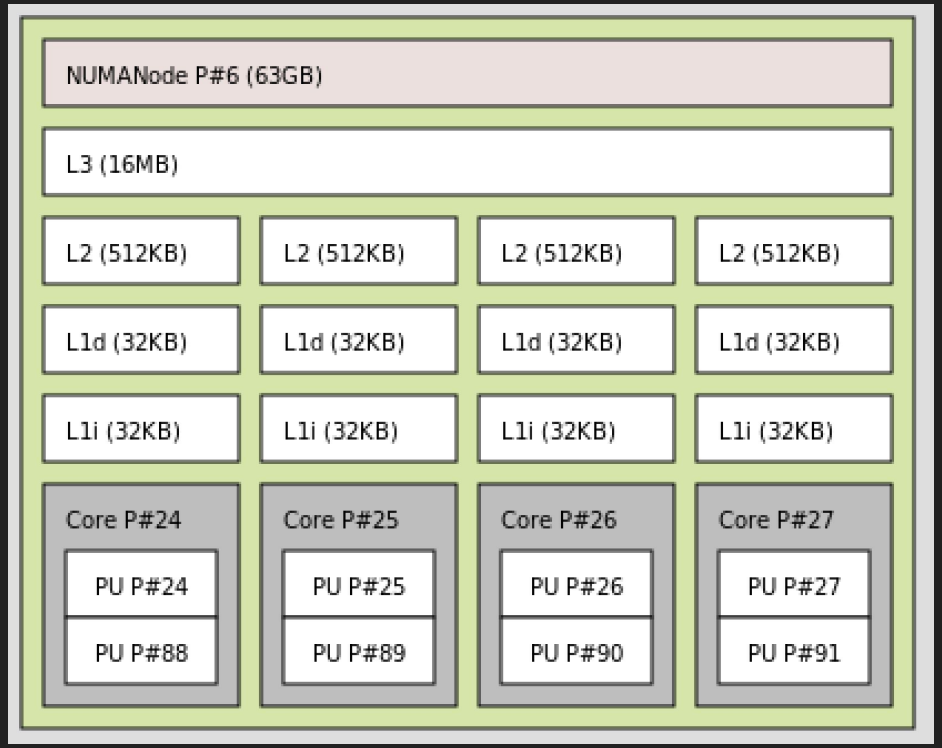
Before we can talk about networking, we need to talk about CPU caches.

THE MEMORY HIERARCHY



```
lstopo --of png > blah.png
```

A Zen2 “Core Complex” (CCX)



Two important properties of data access patterns to remember to maximize the benefit of CPU cache

Spatial locality

Temporal locality

Classic example:

2-D array access in C

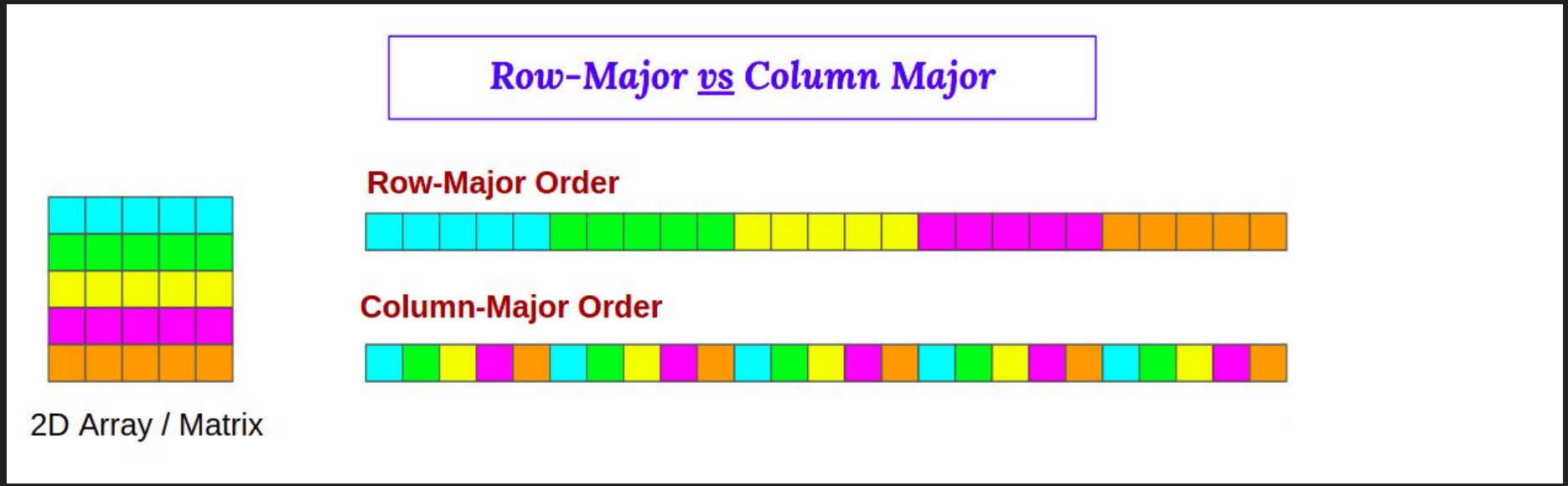
Row major vs column major

Row major order

```
for  (i = 0; i < ARR_SIZE; i++) {
        for (j = 0; j < ARR_SIZE; j++) {
                sum += arr[i][j];
        }
}
```

Column major order

```
for  (i = 0; i < ARR_SIZE; i++) {
        for (j = 0; j < ARR_SIZE; j++) {
                sum += arr[j][i];
        }
}
```



Each color change is a cache miss

Benchmarks

Tinymembench - lots of memory related benchmarks

Running a microbenchmark

Result

```
jdamato@                    :~$ taskset -ac 23 ./ct
beginning row major order test
row major order test complete: 46078ms


beginning column major order test
column major order test complete: 94860ms
```

Result

Row major access is >2x faster
Because data accesses have:

- Spatial locality (accessing nearby data)
- Temporal locality (accessing data again soon)

But: what if we know we don’t need temporal locality?

It’d be nice to avoid the CPU cache evicting in-use data in exchange for data we *know* we won’t touch again

The CPU cache is thrashed unnecessarily.

Useful data is evicted from the cache in exchange for data where there’s no locality benefit.

Using “non-temporal” stores we can write data directly to RAM *without* disturbing the CPU cache.

MOVNTDQA—Load Double Quadword Non-Temporal Aligned Hint

MOVNTQ—Store of Quadword Using Non-Temporal Hint

MOVNTI—Store Doubleword Using Non-Temporal Hint

OK.

What does this have to do with networking?

```
jdamato:~\ $ ethtool -k eth4 | grep nocache
tx-nocache-copy: off
```

ethtool allows you to enable “no cache copy” from userspace on transmit

What does that mean?

Diagram of tx-nocache-copy off

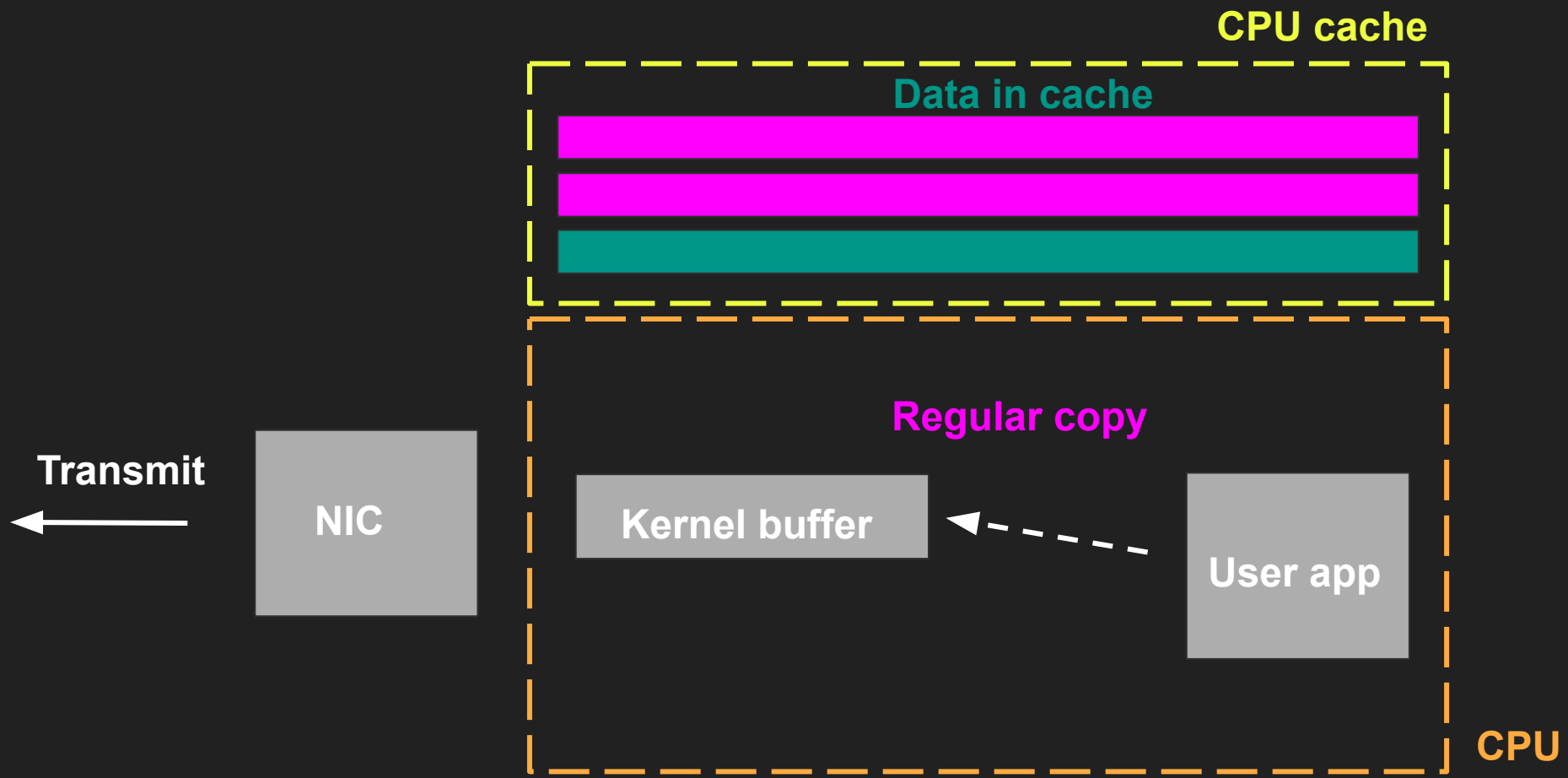
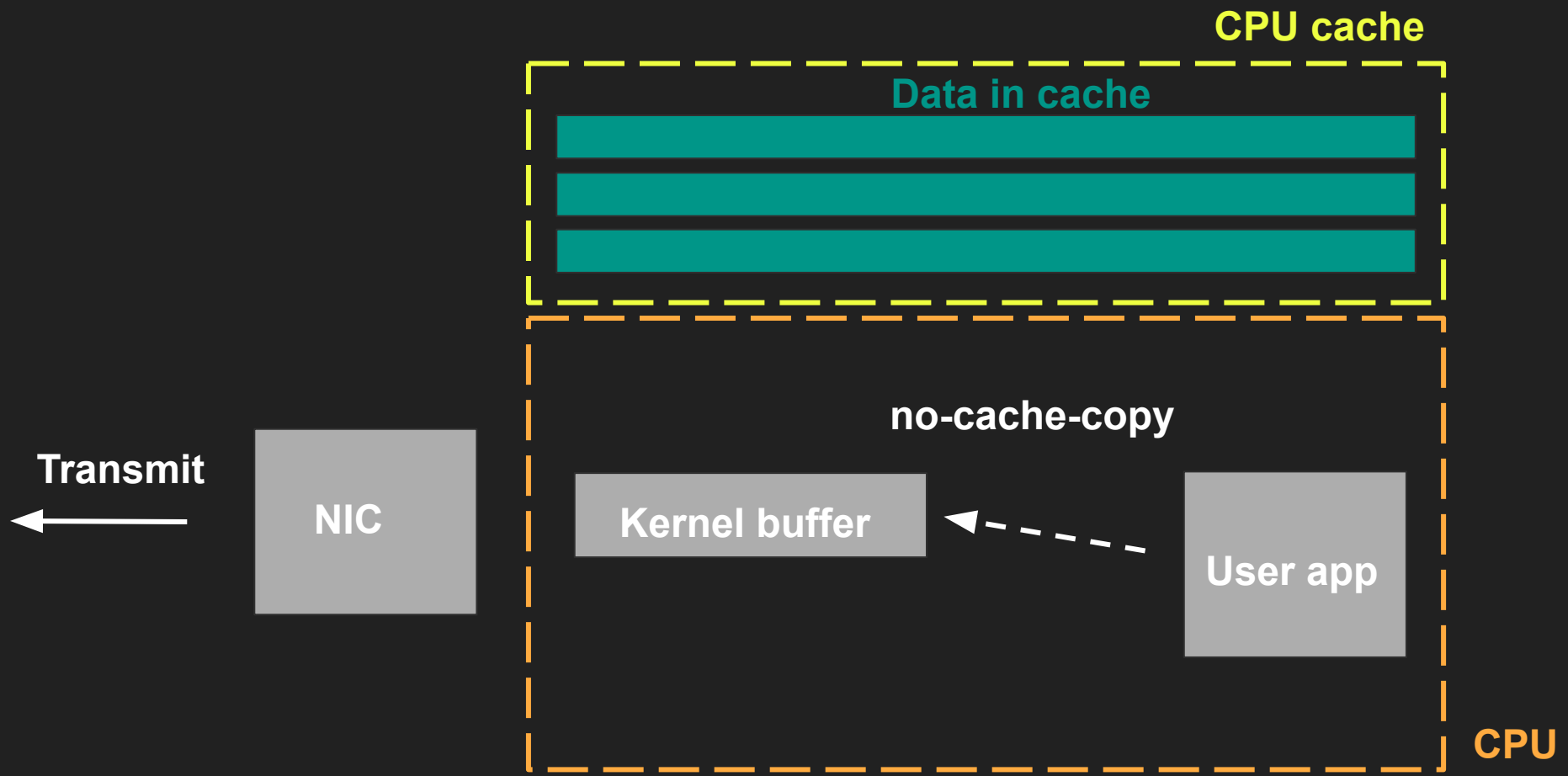

CPU cache
Data in cache
Regular copy
Transmit
NIC
Kernel buffer
User app
CPU

Diagram of tx-nocache-copy **on**


CPU cache
Data in cache
no-cache-copy
Transmit
NIC
Kernel buffer
User app
CPU

With tx-nocache-copy enabled and assuming the kernel or driver don’t touch the data…

The CPU cache is not disturbed.

This can lead to a performance increase:

1. Important application data is kept in the cache
2. Reduction in memory bandwidth consumption

```
commit c6e1a0d12ca7b4f22c58e55a16beacfb7d3d8462
Author: Tom Herbert <therbert@google.com>
Date:   Mon Apr 4 22:30:30 2011 -0700

    net: Allow no-cache copy from user on transmit

    This patch uses __copy_from_user_nocache on transmit to bypass data
    cache for a performance improvement.  skb_add_data_nocache and
    skb_copy_to_page_nocache can be called by sendmsg functions to use
    this feature, initial support is in tcp_sendmsg.  This functionality is
    configurable per device using ethtool.
```

Using 14000 byte request and response sizes demonstrate the effects more dramatically:

No-cache copy disabled:
   79571 tps, 34.34 %utlization
   50/90/95% latency 1584.46 2319.59 5001.76

No-cache copy enabled:
   83856 tps, 34.81% utilization
   50/90/95% latency 2508.42 2622.62 2735.88

Note especially the effect on latency tail (95th percentile).

This seems to provide a nice performance improvement and is consistent in the tests I ran. Presumably, this would provide the greatest benfits in the presence of an application workload stressing the cache and a lot of transmit data happening.

The original author noted a ~45% improvement in p95 times - cool!

BUT…

But:

- It only affects TCP
- It’s interface wide, so it affects every connection
- Seems undesirable in some situations (like software kTLS)

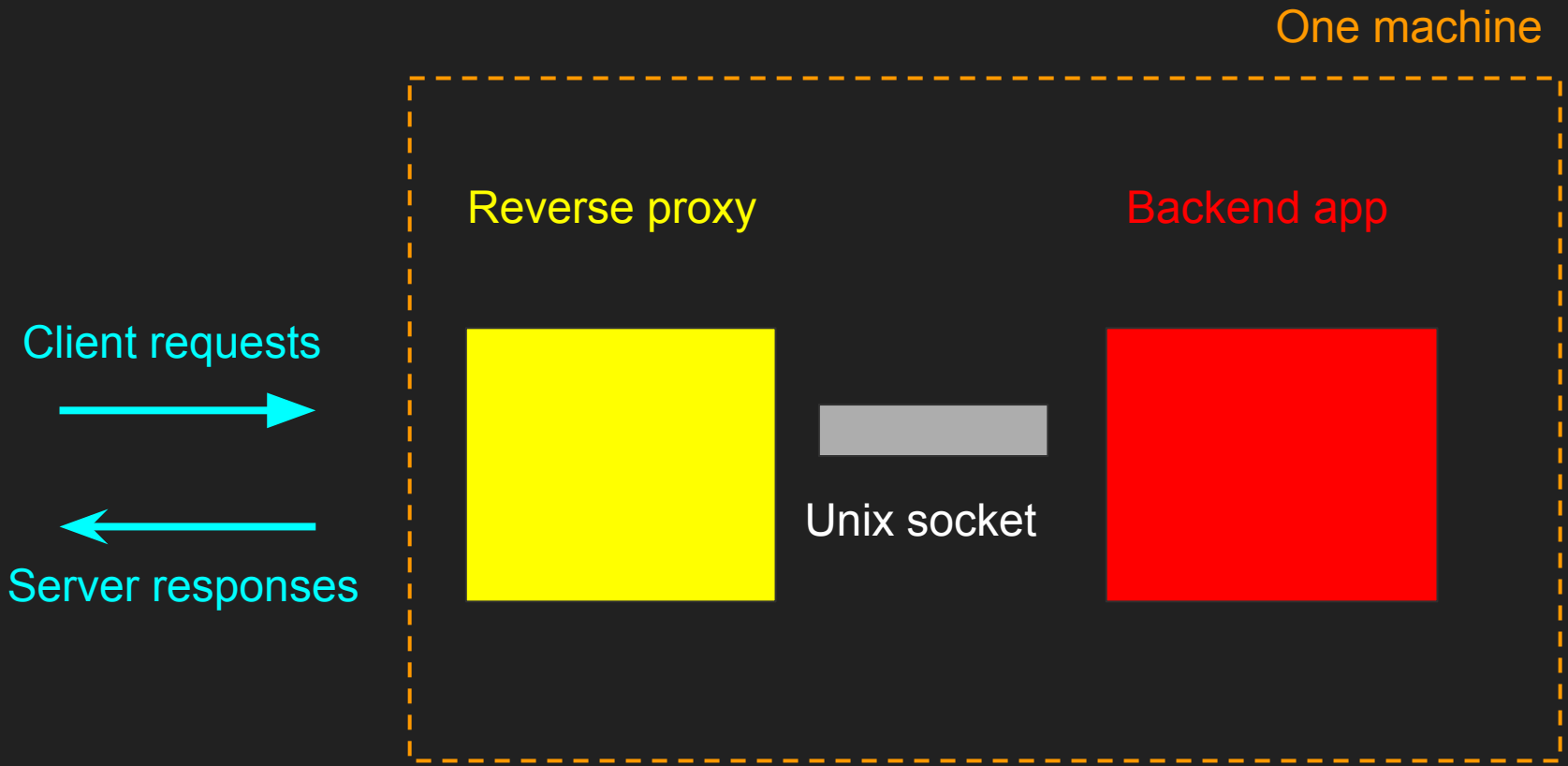
AND…

Other:

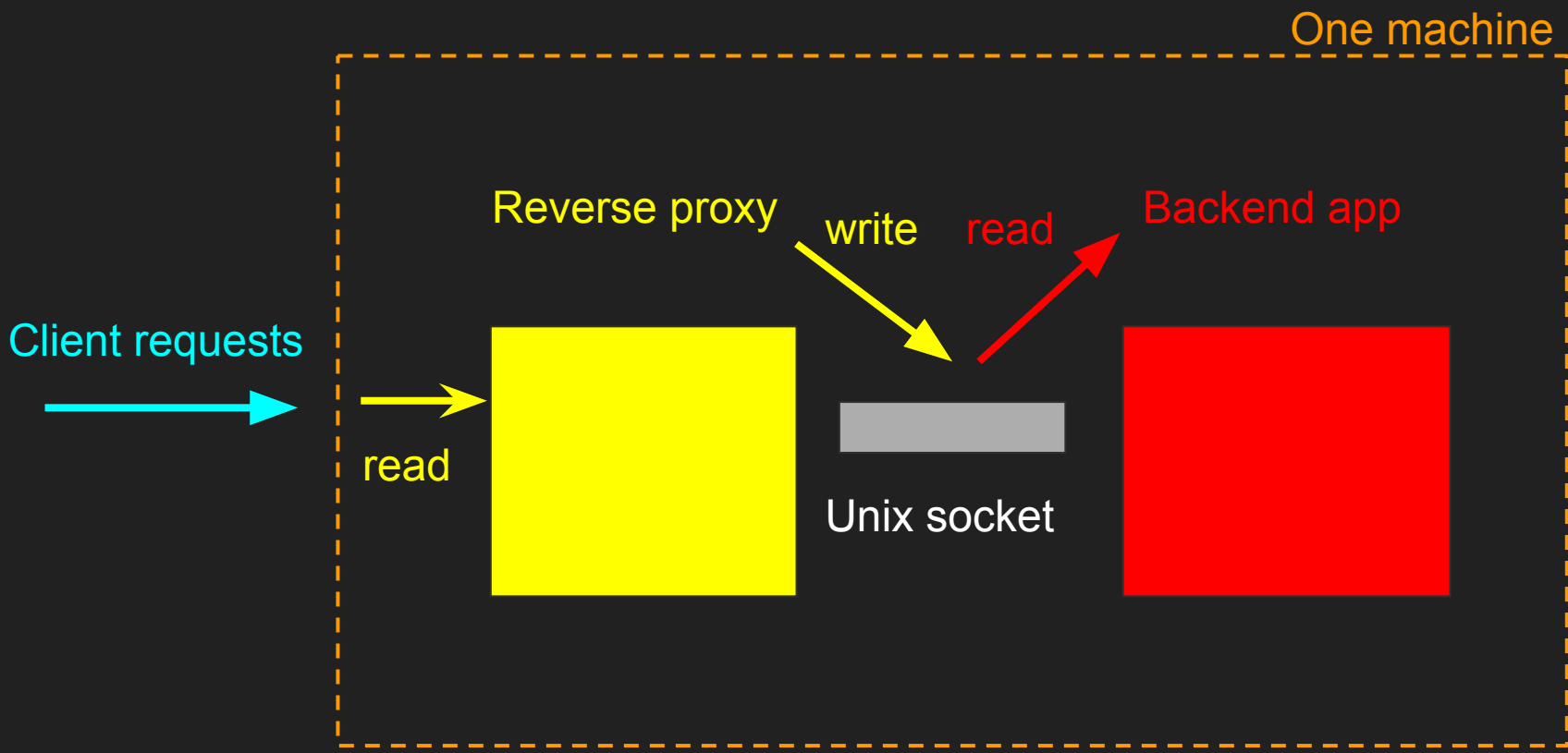
- hardware
- kernel features
- application architectures
- protocols
- address families

Invite interesting use cases which the existing mechanism doesn’t support

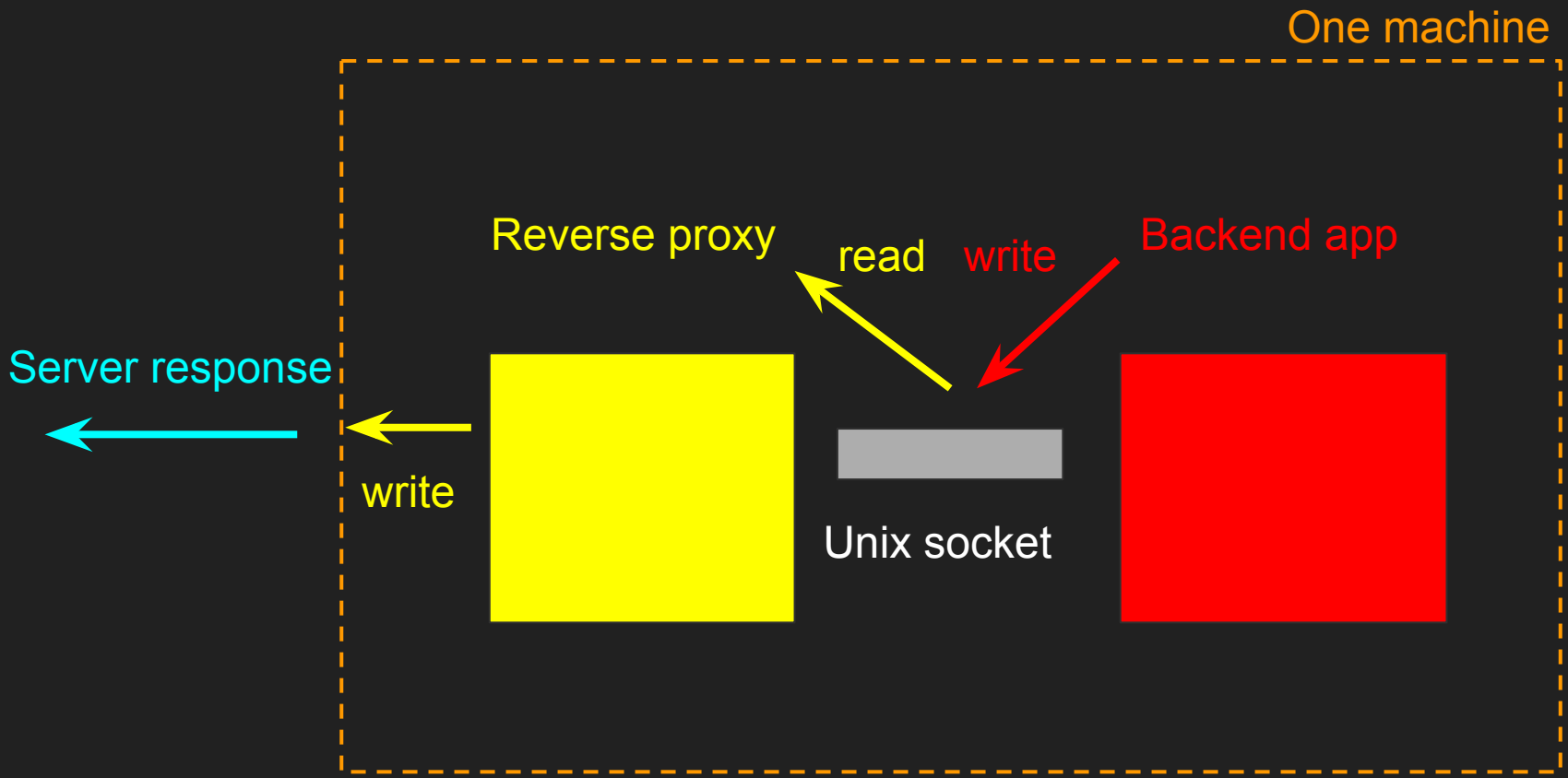
An example of a common app architecture pattern


One machine
Reverse proxy
Backend app
Client requests
Unix socket
Server responses

What typically happens is…


One machine
Reverse proxy
write
read
Backend app
Client requests
read
Unix socket

Backend app does some processing and generates the response….


One machine
Reverse proxy
read
write
Backend app
Server response
write
Unix socket

With lots of buffers being read and written

And thus lots of CPU cache being churned

Even for plain text HTTP, where the proxy doesn’t do much

Is there a way to do this with:

- Less data movement (thus less cache churn)
- For specific connections
- For more than just TCP

What if:

sendmsg was extended to allow the *application* to decide when a non-temporal (NT) write is needed?

I submit an RFC to add a new sendmsg flag, MSG_NTCOPY.

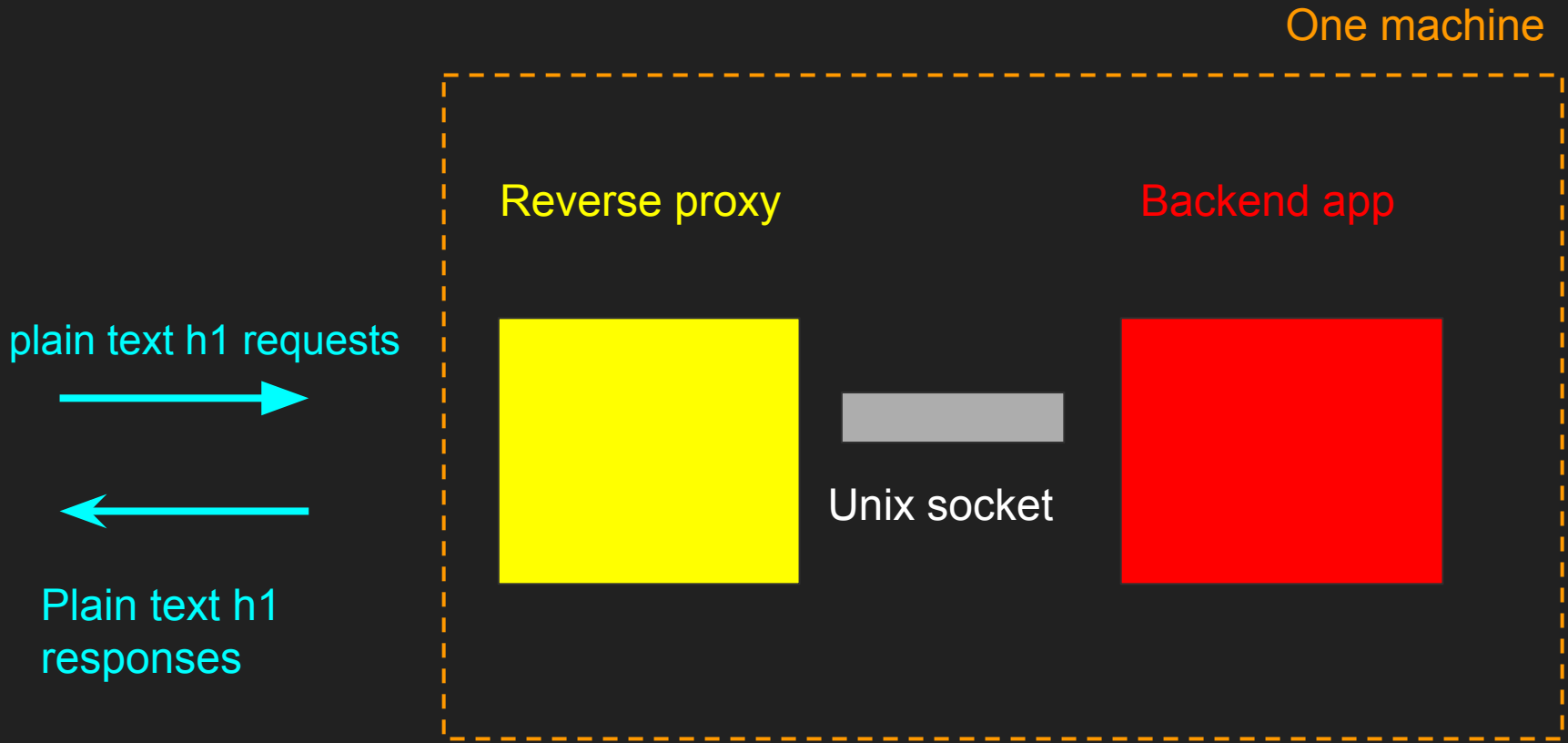
```
From: Joe Damato <jdamato@fastly.com>
To: x86@kernel.org, Alexander Viro <viro@zeniv.linux.org.uk>,
        Borislav Petkov <bp@alien8.de>,
        Dave Hansen <dave.hansen@linux.intel.com>,
        David Ahern <dsahern@kernel.org>,
        "David S. Miller" <davem@davemloft.net>,
        Eric Dumazet <edumazet@google.com>,
        Hideaki YOSHIFUJI <yoshfuji@linux-ipv6.org>,
        "H. Peter Anvin" <hpa@zytor.com>, Ingo Molnar <mingo@redhat.com>,
        Jakub Kicinski <kuba@kernel.org>,
        linux-kernel@vger.kernel.org, netdev@vger.kernel.org,
        Paolo Abeni <pabeni@redhat.com>,
        Thomas Gleixner <tglx@linutronix.de>
Cc: Joe Damato <jdamato@fastly.com>
Subject: [RFC,net-next,x86 v2 0/8] Nontemporal copies in sendmsg path
Date: Sun, 12 Jun 2022 01:57:49 -0700   [thread overview]
Message-ID: <1655024280-23827-1-git-send-email-jdamato@fastly.com> (raw)

Greetings:

Welcome to RFC v2.

This is my first series that touches more than 1 subsystem; hope I got the
various subject lines and to/cc-lists correct.

Based on the feedback on RFC v1 [1], I've made a few changes:

- Removed the indirect calls.
- Simplified the code a bit by pushing logic down to a wrapper around
  copyin.
- Added support for the 'MSG_NTCOPY' flag to udp, udp-lite, tcp, and unix.
```

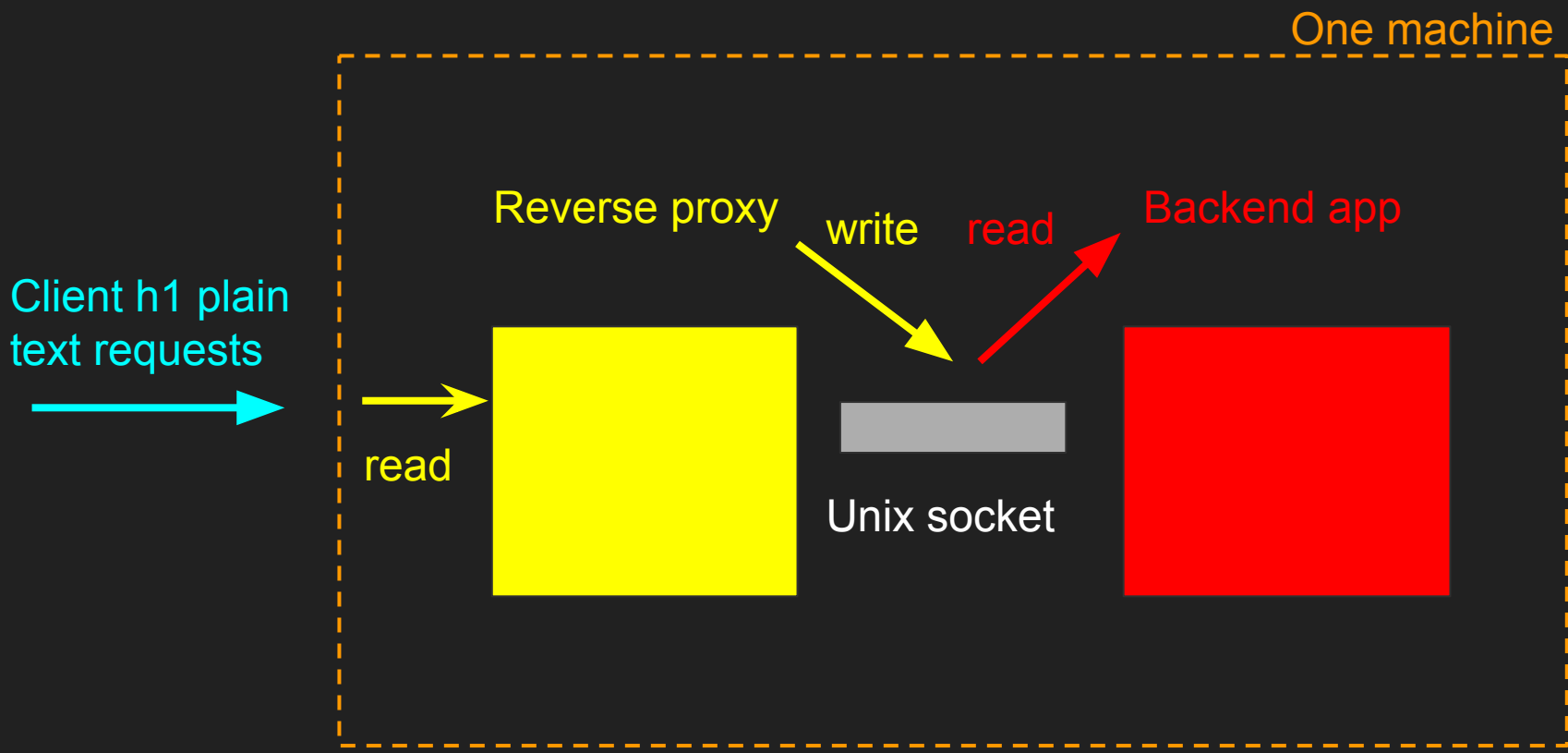
The idea is:

1. User apps know when they will or won't need temporal locality
2. Support more than just TCP
3. Allow for preservation of CPU cache contents in common app architecture patterns

Let’s look at 1 specific use case with disproportionate benefit:

Plain text http1

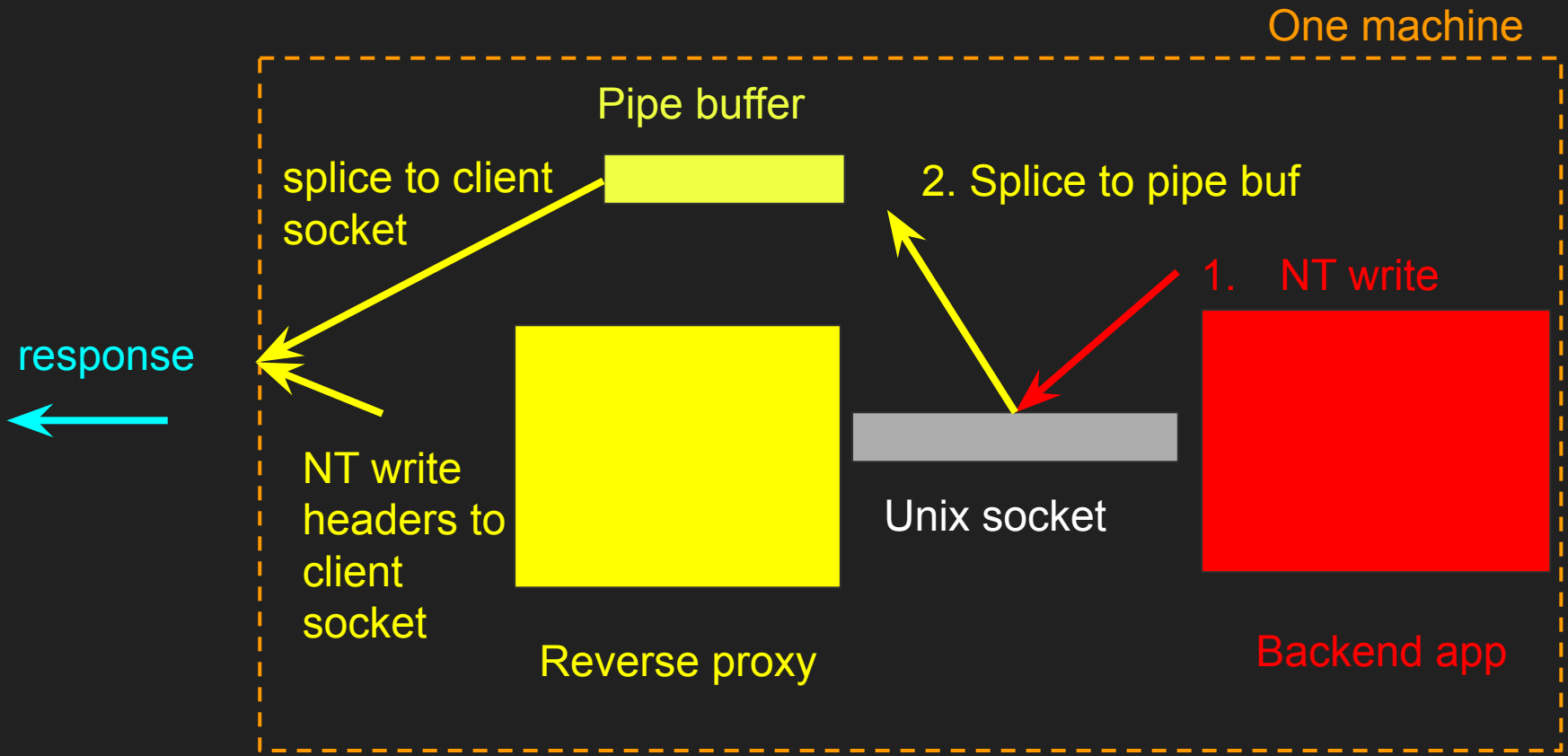

One machine
Reverse proxy
Backend app
plain text h1 requests
Unix socket
Plain text h1
responses


One machine
Reverse proxy
write
read
Backend app
Client h1 plain
text requests
read
Unix socket

The backend app “knows” the request is plain text h1

Using the RFC, we can avoid dirtying the cache and save memory bandwidth:

sendmsg(unix_fd, msg, MSG_NTCOPY)


One machine
Pipe buffer
splice to client socket
2. Splice to pipe buf
1. NT write
response
NT write headers to client socket
Unix socket
Reverse proxy
Backend app

1. The backend “knows” the request is plain text
2. Backend uses a non-temporal write to the unix socket
3. Reverse proxy uses:
    a. Non-temporal write for headers
    b. splice for the body

And so ….

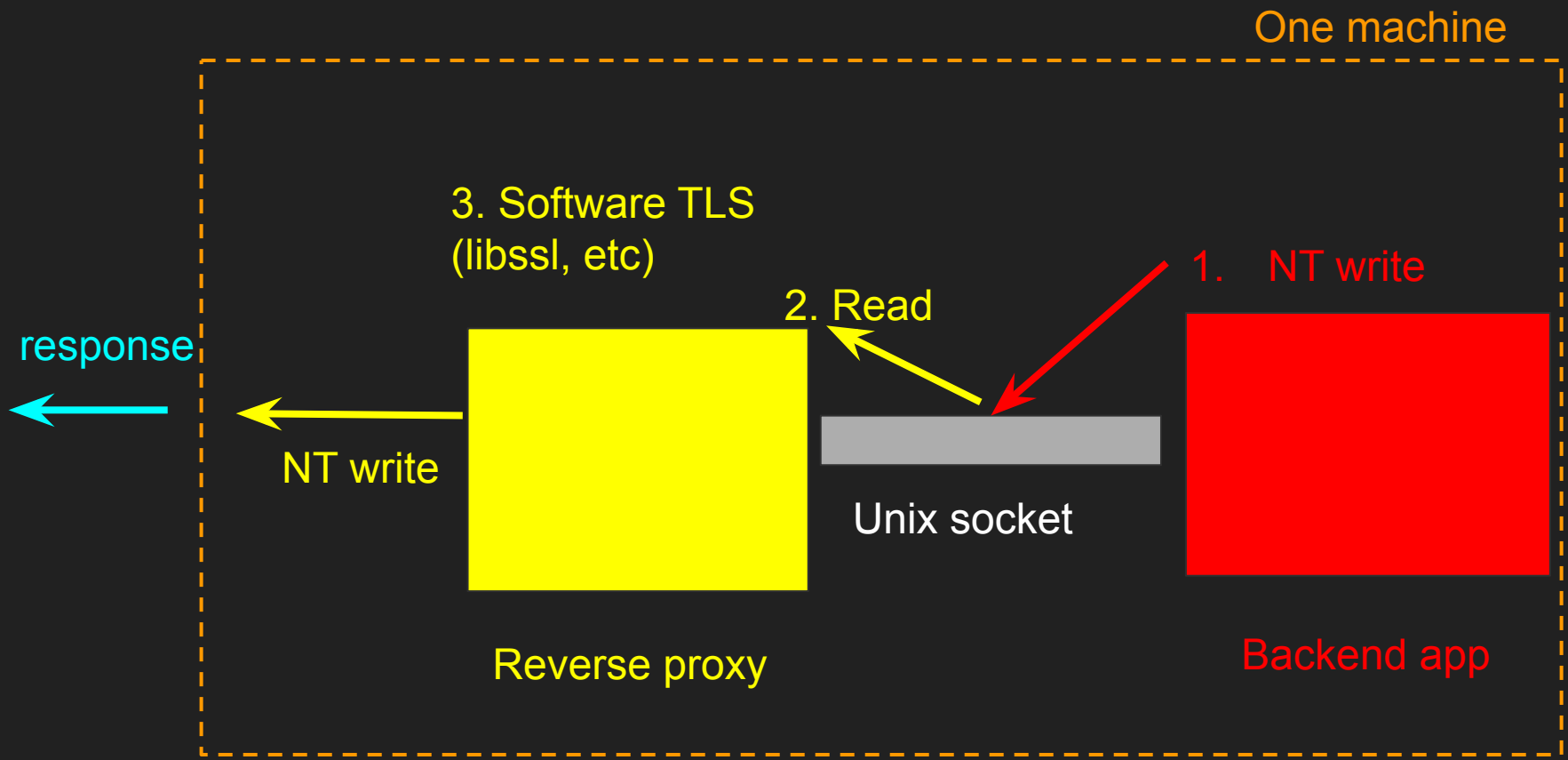
In this particular scenario, we avoid disrupting the cache:

1. NT-write from backend to unix socket
2. NT-write from reverse proxy to client
3. splice the body from pipebuf to client

The entire time the CPU cache remains undisturbed as much as possible.

Hot application data stays resident.

OK what about TLS ?


One machine
3. Software TLS
(libssl, etc)
2. Read
1. NT write
response
NT write
Unix socket
Reverse proxy
Backend app

In this case, the backend app preserves the CPU cache as much as it can.

Reverse proxy *wants* the data in cache so that TLS will be fast

Still can do an NT-write to output.

Many other cases where NT flexibility is helpful:

- kTLS (hardware offloaded vs software)
- QUIC (UDP)

Benchmark data !

Microbenchmark

- Based on existing unix_thr benchmark
- fork():
  - parent does NT write to unix socket
  - child does splice to pipebuf + splice to /dev/null
- Lots of command line flags to benchmark different sizes

From the RFC

```
Test system: AMD EPYC 7662 64-Core Processor,
             64 cores / 128 threads,
             512kb L2 per core shared by sibling CPUs,
             16mb L3 per NUMA zone,
             AMD specific settings: NPS=1 and L3 as NUMA enabled

Test: 1048576 byte object,
      100,000 iterations,
      512kb pipe buffer size,
      512kb unix socket send buffer size
```

From the RFC

```
Test pinned to CPUs 1 and 2 which do *not* share an L2 cache, but do share
an L3.

Command line for "normal" copy:
% time taskset -ac 1,2 ./unix-nt-bench 1048576 100000 0 524288 524288

Mode                    real time (sec.)                throughput (Mb/s)
"Normal" copy           10.630                          78,928
MSG_NTCOPY              7.429                           112,935
```

(~43% increase in throughput!)

From the RFC

```
Same test as above, but pinned to CPUs 1 and 65 which share an L2 (512kb)
and L3 cache (16mb).

Command line for "normal" copy:
% time taskset -ac 1,65 ./unix-nt-bench 1048576 100000 0 524288 524288

Mode                    real time (sec.)                throughput (Mb/s)
"Normal" copy           12.532                          66,941
MSG_NTCOPY              9.445                           88,826
```

(~33% increase in throughput!)

These numbers definitely seem bizarre so it’s not surprising that I got this response on the mailing list….

From: Al Viro <viro@zeniv.linux.org.uk>
To: Joe Damato <jdamato@fastly.com>
Cc: linux-kernel@vger.kernel.org

Just a sanity check - your testing is *not* with KASAN/KCSAN, right?

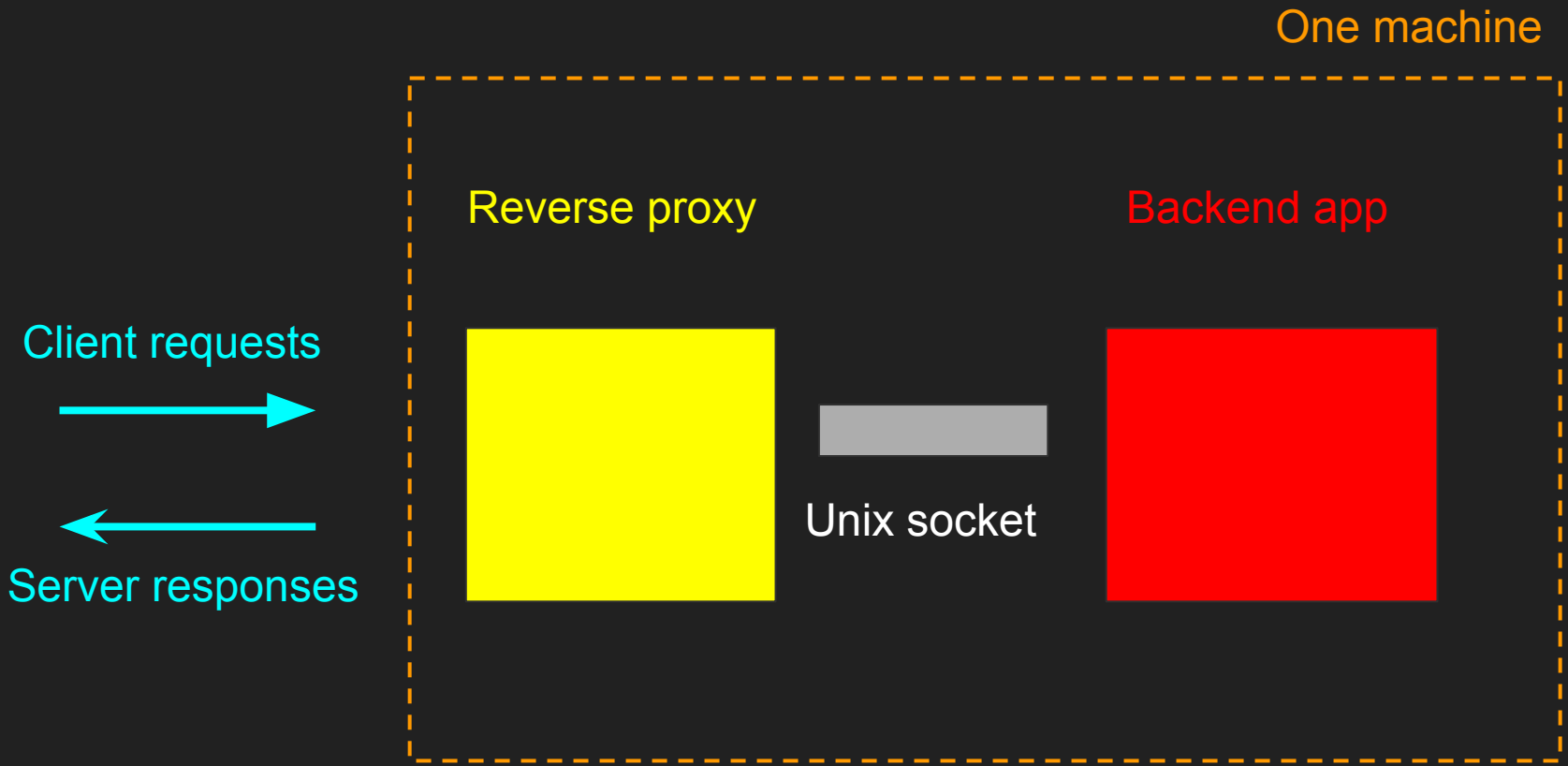
And BTW, why is that only on the userland side? If you are doing that at all, it would make sense to cover the memcpy() side as well...

(We’ll come back to the memcpy bit in a moment)

Micro benchmarks show a huge improvement.

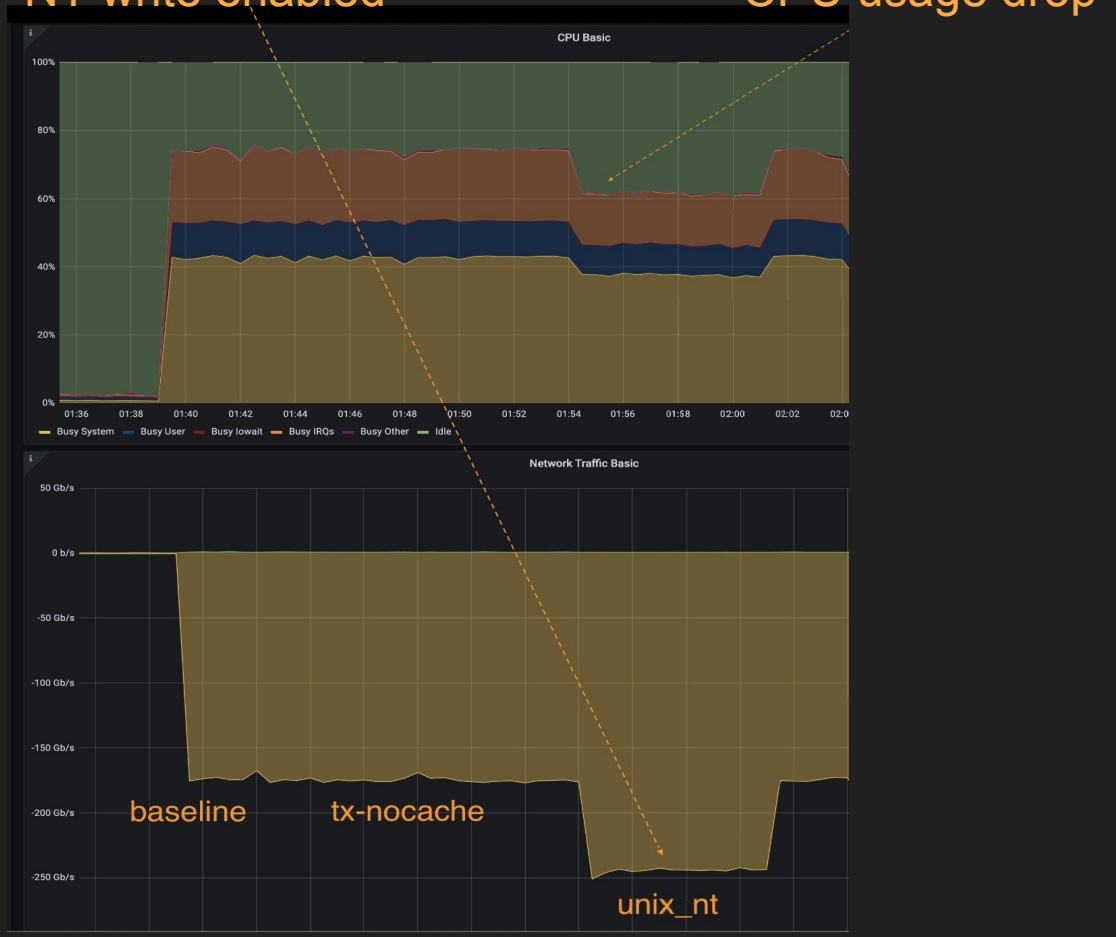
What about an HTTP workload benchmark?

HTTP benchmark uses reverse proxy + backend architecture pattern


One machine
Reverse proxy
Backend app
Client requests
Unix socket
Server responses

Sorry I know this is hard to see, but:

- Plain text http 1
- Reverse proxy + unix socket + backend
- 170 gbps in “normal” copy mode
- 245 gbps in when NT copy mode enabled
- And a CPU usage drop



tx-nocache-copy ON

Doesn’t help in this benchmark because the reverse proxy is *already using splice* (and not touching the data on TX).

Using sendmsg MSG_NTCOPY does help

An NT copy into the unix socket preserves the cache and generates less memory bandwidth traffic

170 gbps normal copy
245 gbps NT copy

~44% increase in throughput

What about DDIO (Intel CPUs) ?

DDIO allows DMA directly from cache. Thus, non-temporal writes may not be useful if buffers are sized properly.

This is CPU and application specific. Users should be aware that DDIO will affect their results.

If your CPU (Intel) has DDIO support, I strongly recommend reading this paper to learn more about networking performance with DDIO.

So if this is true in the kernel:

what about memcpy in userland?

(which is what I *think* Al Viro was asking?)

glibc tunable for memcpy/memmove

[PATCH 0/1] x86: Tuning NT Threshold parameter for AMD machines

Sajan Karumanchi sajan.karumanchi@gmail.com
Wed Aug 19 10:45:38 GMT 2020

- Previous message (by thread): Use MPC 1.2.0 in build-many-glibcs.py
- Next message (by thread): [PATCH 1/1] x86: Tuning NT Threshold parameter for AMD machines.
- **Messages sorted by:** [date] [thread] [subject] [author]

```
Tuning NT threshold parameter '__x86_shared_non_temporal_threshold' to 2/3 of
shared cache size on AMD Zen[1|2] machines brings in performance gains
for memcpy/memmove as per the Large and Walk Bench variant reuslts.

As there are run to run variations in bench results, I took average of 100 runs
for both vanilla and patched glibc.

AMD ZEN[1/2] architectures doesn't have ERMS cpu feature.
So, on ZEN architecutre memcpy takes 'memcpy_avx_unaligned' entry point.

Below is the large bench test results comparision for entry points:
avx_unaligned and avx_unaligned_erms.
```

glibc tunable for memcpy/memmove

[PATCH 1/1] x86: Optimizing memcpy for AMD Zen architecture.

sajan.karumanchi@amd.com sajan.karumanchi@amd.com
Thu Oct 22 04:50:05 GMT 2020

- Previous message (by thread): [PATCH 0/1] Optimizing memcpy for AMD Zen architecture.
- Next message (by thread): [PATCH 1/1] x86: Optimizing memcpy for AMD Zen architecture.
- **Messages sorted by:** [date] [thread] [subject] [author]

```
From: Sajan Karumanchi <sajan.karumanchi@amd.com>

Modifying the shareable cache '__x86_shared_cache_size', which is a
factor in computing the non-temporal threshold parameter
'__x86_shared_non_temporal_threshold' to optimize memcpy for AMD Zen
architectures.
In the existing implementation, the shareable cache is computed as 'L3
per thread, L2 per core'. Recomputing this shareable cache as 'L3 per
CCX(Core-Complex)' has brought in performance gains.
As per the large bench variant results, this patch also addresses the
regression problem on AMD Zen architectures.

Reviewed-by: Premachandra Mallappa <premachandra.mallappa@amd.com>
Signed-off-by: Premachandra Mallappa <premachandra.mallappa@amd.com>
Signed-off-by: Sajan Karumanchi <sajan.karumanchi@amd.com>
```

More recently, Noah Goldstein from Intel has been investigating NT writes for memset

glibc tunable for NT memset (intel only)

[v1] x86: Improve large memset perf with non-temporal stores [RHEL-29312]

1936755 | diff | mbox | series

Message ID 20240519004347.2759850-1-goldstein.w.n@gmail.com
State New
Headers show
Series [v1] x86: Improve large memset perf with non-temporal stores [RHEL-29312] | expand

Commit Message

Noah Goldstein — May 19, 2024, 12:43 a.m. UTC

```
Previously we use `rep stosb` for all medium/large memsets. This is
notably worse than non-temporal stores for large (above a
few MBs) memsets.
See:
https://docs.google.com/spreadsheets/d/1opzukzvum4n6-RUVHTGddV6RjAEil4P2uMjjQGLbLcU/edit?usp=sharing
For data using different stategies for large memset on ICX and SKX.

Using non-temporal stores can be up to 3x faster on ICX and 2x faster
on SKX. Historically, these numbers would not have been so good
because of the zero-over-zero writeback optimization that `rep stosb`
is able to do. But, the zero-over-zero writeback optimization has been
removed as a potential side-channel attack, so there is no longer any
good reason to only rely on `rep stosb` for large memsets. On the flip
size, non-temporal writes can avoid data in their RFO requests saving
```

glibc tunable for NT memset (AMD support)

x86: Enable non-temporal memset tunable for AMD

Message ID	20240607230447.52478-1-jdamato@fastly.com
State	Committed
Commit	bef2a827a55fc759693ccc5b0f614353b8ad712d
Headers	show
Series	x86: Enable non-temporal memset tunable for AMD | expand

Checks

Context	Check	Description
redhat-pt-bot/TryBot-apply_patch	success	Patch applied to master at the time it was sent
redhat-pt-bot/TryBot-32bit	success	Build for i686
linaro-tcwg-bot/tcwg_glibc_build--master-aarch64	success	Testing passed
linaro-tcwg-bot/tcwg_glibc_build--master-arm	success	Testing passed
linaro-tcwg-bot/tcwg_glibc_check--master-arm	success	Testing passed
linaro-tcwg-bot/tcwg_glibc_check--master-aarch64	success	Testing passed

Commit Message

Joe Damato June 7, 2024, 11:04 p.m. UTC

```
  In commit 46b5e98ef6f1 ("x86: Add seperate non-temporal tunable for
memset") a tunable threshold for enabling non-temporal memset was added,
but only for Intel hardware.

Since that commit, new benchmark results suggest that non-temporal
memset is beneficial on AMD, as well, so allow this tunable to be set
for AMD.

See:
https://docs.google.com/spreadsheets/d/1opzukzvum4n6-RUVHTGddV6RjAEil4P2uMjjQGLbLcU/edit?usp=sharing
which has been updated to include data using different stategies for
large memset on AMD Zen2, Zen3, and Zen4.
```

If you are super curious, Noah has been compiling a spreadsheet of memset performance across many different CPUs.

In conclusion:

- tx-nocache-copy is useful for a very narrow set of applications
- Applications “know” when they don’t need temporal locality
- glibc has tunables for NT writes in memcpy and memset
- Microbenchmarks and http benchmarks show a performance boost

Maybe:

sendmsg should be extended to allow user applications to decide if their writes should or should not have temporal locality?