NX-GZIP hardware acceleration on Talos II POWER9

2026-05-15T00:00:00+02:00

POWER9 processors[1] ship with two hardware compression engines: NX-842 (a kernel crypto API accelerator for the 842 algorithm) and NX-GZIP (a gzip/deflate-compatible engine accessible from userspace). NX-GZIP is the interesting one: it provides zlib-compatible acceleration via `LD_PRELOAD` without rebuilding any software. This post documents a possible setup on a Talos II running Arch Linux ppc64le (kernel 6.19.11) and shows some simple benchmarks to get a feel on what to expect with respect to performance improvement.

Background

POWER9 exposes the compression hardware through the VAS (Virtual Accelerator Switchboard) subsystem[2], there are two engines:

- **NX-842**: 842 algorithm, registered in the kernel crypto API. Loads automatically once `nx-compress-powernv.ko` is active. No userspace setup required.[3]
- **NX-GZIP**: gzip/deflate engine, exposed as `/dev/crypto/nx-gzip`. Requires a one-time NVRAM flag and the libnxz userspace library.[4]

The Arch Linux ppc64le kernel has `CONFIG_PPC_VAS=y` and `CONFIG_CRYPTO_DEV_NX=y` already set — no kernel rebuild is needed.

Step 1: Enable VAS userspace access in NVRAM

To enable exposure of the engine, the skiboot firmware reads the `vas-user-space` NVRAM option at boot. A kexec (fast reset) is not sufficient — only a full IPL picks up NVRAM changes.

[1] Wikipedia: POWER9
[2] kernel.org: VAS userspace API
[3] power-gzip wiki: NX-842 compression accelerator
[4] kernel.org: VAS userspace API

```
sudo nvram --update-config "vas-user-space=enable" --partition ibm,skiboot
# Full power cycle required - kexec will not work

# Verify
sudo nvram -p ibm,skiboot --print-config
# "ibm,skiboot" Partition
# ---------------------------
# vas-user-space=enable
```

After reboot, verify in the OPAL message log:

```
grep "vas-user-space" /sys/firmware/opal/msglog
# NVRAM: Searched for 'vas-user-space' found 'enable'
# VAS: Initialized chip 0 / VAS: Initialized chip 8
# NX0: gzip Coprocessor Enabled / NX8: gzip Coprocessor Enabled
```

Step 2: Verify `/dev/crypto/nx-gzip` exists

With $\text{CONFIG}_{\text{PPCVAS}}$ and $\text{CONFIG}_{\text{CRYPTODEVNX}}$ enabled (both present in the Arch ppc64le kernel), the device node is created automatically by the VAS subsystem on boot:

```
ls -la /dev/crypto/nx-gzip
# crw------- 1 root root 236, 0 ...
```

Step 3: Create system group and udev rule

To make usage of the device a bit easier, we assign it a group and proper permissions. udev requires a **system group** (GID < 1000) for device node rules — a regular user group silently fails with "Not a system group" and the rule is ignored.

```
sudo groupadd --system nx-gzip
sudo usermod -aG nx-gzip $USER
echo 'KERNEL=="nx-gzip", GROUP="nx-gzip", MODE="0660"' | \
  sudo tee /etc/udev/rules.d/99-nx-gzip.rules
sudo udevadm control --reload
sudo udevadm trigger --subsystem-match=nx-gzip
```

Verify:

```
sudo udevadm test /devices/virtual/nx-gzip/nx-gzip 2>&1 | grep -E "GROUP|MODE"
# GROUP="nx-gzip": Set group ID: 939
# MODE="0660":    Set mode: 0660
```

Users in the nx-gzip group can now use the crypto device. Log out and back in (or use `newgrp nx-gzip`) for group membership to take effect.

Step 4: Build libnxz

The library that catches calls to libz and reroutes them to the hardware lives at github.com/libnxz/power-gzip. Clone it.

There is onne build fix needed in `lib/nx_gzlib.c` line 140 declares `digit` as `char*` but `strpbrk` returns `const char*`, which gcc -Werror rejects:

```
--- a/lib/nx_gzlib.c
+++ b/lib/nx_gzlib.c
@@ -140 +140 @@
-   char* digit;
+   const char* digit;
```

If you do not apply the patch manually, these lines will give you a working library:

```
cd ~/dat/src/power-gzip
./configure
sed -i 's/\tchar\* digit;/\tconst char* digit;/' lib/nx_gzlib.c
make -j$(nproc) -C lib
# Produces: lib/.libs/libnxz.so.0.0.65
```

Step 5: Use via $\text{LD}_{\text{PRELOAD}}$

libnxz intercepts zlib API calls (`compress`, `deflate`, `inflate`, etc.) from programs that dynamically link `libz.so`. It does **not** work with the `gzip` binary — GNU gzip bundles its own deflate and never calls `libz.so`.

The generic use of the library, for programs that use libz, is:

```
LD_PRELOAD=<path-to>/libnxz.so <program>
```

To verify hardware is actually used:

```
# Run it
NX_GZIP_LOGFILE=/tmp/nx.log NX_GZIP_VERBOSE=2 NX_GZIP_TRACE=8 LD_PRELOAD=.../libnxz.so <prog

# Verify use
grep "deflate(nx)" /tmp/nx.log    # count should be > 0
```

This works with `python3`, via the `zlib` module, `rsync`, `pigz`, Java, and any program that dynamically links `libz.so`.

Does **not** work with: `gzip`, `zstd`, `lz4` — these have their own compression implementations.

Before wrapping a binary, check that it does link `libz.so` and not `libz-ng.so.2` (zlib-ng):

```
ldd /usr/bin/someprogram | grep libz
# Must show libz.so.1 - not libz-ng.so.2
```

For example, `git` on my machine links `libz-ng.so.2`, so libnxz cannot intercept it — zero NX operations will be recorded.

Benchmarks

Benchmark: Python's `zlib.compress()` with level 1, comparing software zlib against NX-GZIP under `LD_PRELOAD`.

```
import zlib, os, time

# Random data (incompressible - worst case)
data = os.urandom(1024 * 1024) * 50    # 50 MB
t = time.perf_counter()
out = zlib.compress(data, 1)
print(f'{len(data) / (time.perf_counter() - t) / 1024**2:.0f} MB/s')

# Compressible data
data = (b'Hello world this is some compressible text data ' * 64) * (1024 * 32)    # 96 MB
t = time.perf_counter()
out = zlib.compress(data, 1)
print(f'{len(data) / (time.perf_counter() - t) / 1024**2:.0f} MB/s')
```

NX-GZIP results (zlib level 1, POWER9 DD2.2, 2 sockets / 8 cores):

Input type	Size	Software zlib	NX-GZIP	Speedup
Random data	50 MB	24 MB/s	710 MB/s	~29×
Compressible	96 MB	355 MB/s	6000 MB/s	~17×

NX-842 (kernel crypto API, separate engine, via `nx-compress-powernv.ko`):

Path	Throughput	Speedup
Software 842 fallback	~104 MB/s	
NX-842 hardware	~11000 MB/s	~106×

It's an artificial benchmark, but the benefit is pretty clear.

Real-world use: mksquashfs (270 MB source tree, 32 threads)

Input: the skiboot git repository (~270 MB of source code).

Variant	Wall time	User CPU	CPU%	Output size
Software	0.516s	7.87s	1561%	63.8 MB
NX-GZIP	0.102s	0.22s	522%	67.2 MB
Speedup	~5×	~36×		-5% ratio

NX trades ~5% compression ratio for 5× wall-time and 36× CPU reduction. At 32 threads the software path saturates all cores; NX offloads deflate to the coprocessor and frees CPU for other work. `ERR_NX_TARGET_SPACE` retries in the verbose log are normal for highly compressible data — the engine splits chunks, not a software fallback.

Example Wrapper scripts

To use, say with curl, the accelerated compression a simple wrapper script early in your path is sufficient. A thin wrapper script in `~/bin/` transparently applies `LD_PRELOAD` for curl that links `libz.so`:

```
#!/bin/sh
exec env LD_PRELOAD=/usr/local/lib/libnxz.so /usr/bin/curl "$@"
```

Programs verified to link `libz.so` and confirmed working: `curl`, `wget`, `cargo`, `ffmpeg`, `bsdtar`, `mksquashfs`, `unsquashfs`, `qemu-img`, `mariadb-dump`, `pg_dump`, `pg_restore`, `pg_basebackup`.

References

- libnxz wiki: Enable nx-gzip on POWER9
- power-gzip source on GitHub