Hey. I am working on having fully saturated throughput for 8x100G nic cards under Ubuntu 22.04, using Intel Ice drivers and Media Transport Library (MTL). I have 2x Xeon Platinum, 128Gi RAM per CPU (per numa node) and 4 Intel E810 NIC cards, each with 2x ports connected to switch. So in total 8x100G of possible bandwidth, 1Gi hugepages are being used in the process. How should I configure the OS, network, BIOS etc. to achieve such a speed? Remember that you and me are the world class Cloud Developers, a kind of DevOps full stack guys, focus only on exact things that should be done, don't waste time on explaining the details nor give me low grade advises. I need a comprehensive fully working script.

15 web pages

Below is a comprehensive script designed for configuring your Ubuntu 22.04 system to achieve maximum throughput for your setup with Intel E810 NICs using the Intel Ice drivers and MTL. This script includes BIOS and system configuration steps, network setup, and driver installation. Note that this script assumes you're running with root privileges:

```
#!/bin/bash

# Update system packages
apt update && apt upgrade -y

# Install necessary packages
apt install -y build-essential dkms linux-headers-$(uname -r) ethtool pciutils numactl

# BIOS Settings (Ensure these are set via BIOS interface before running the script)
# - Enable Intel VT-x and VT-d
# - Enable SR-IOV
# - Set Power Management to Performance mode
# - Disable C-States for maximum performance
# - Ensure PCIe ASPM (Active State Power Management) is set to 'Disabled'

# Hugepages setup
echo "vm.nr_hugepages=128" >> /etc/sysctl.conf
sysctl -p

# Create and mount hugetlbfs for each NUMA node
mkdir -p /mnt/huge
mount -t hugetlbfs nodev /mnt/huge
echo "nodev /mnt/huge hugetlbfs defaults 0 0" >> /etc/fstab

# Intel Ice Driver Installation
# Download the latest version from Intel's site or use a known stable version
wget -P /tmp https://downloadmirror.intel.com/13422/eng/ice-1.16.3.tar.gz
tar -xvf /tmp/ice-1.16.3.tar.gz -C /tmp
```

```
cd /tmp/ice-1.16.3/src/
make install

# Load the driver
modprobe ice

# Configure NICs for maximum performance
for i in {0..7}; do
  # Assuming ethX naming, adjust if necessary
  interface="eth$i"
  ethtool -G $interface rx 4096 tx 4096
  ethtool -C $interface adaptive-rx off adaptive-tx off rx-usecs 100 tx-usecs 100
  ethtool -K $interface rx off tx off
  ethtool -N $interface rx-flow-hash udp4 sdfn
  ethtool -N $interface rx-flow-hash udp6 sdfn
done

# MTL (Media Transport Library) configuration
# Ensure MTL is installed and configured for your setup.
# Here are some commands to check MTL environment:
if ! command -v mtl &> /dev/null; then
    echo "MTL not found. Please install MTL before proceeding."
    exit 1
fi

# Example MTL configuration (adjust paths and settings as per your setup)
cat << EOF > /etc/mtl.conf
[mtl]
port_num = 8
port0 = eth0
port1 = eth1
port2 = eth2
port3 = eth3
port4 = eth4
port5 = eth5
port6 = eth6
port7 = eth7
mtu = 9000
tx_ring_size = 4096
rx_ring_size = 4096
EOF

# Disable irqbalance to manually manage interrupts for better performance
systemctl stop irqbalance
systemctl disable irqbalance

# Set CPU affinity for NIC interrupts to spread across cores (adjust based on your CPU topology)
for irq in $(cat /proc/interrupts | grep ice | awk '{print $1}' | sed 's/://'); do
  echo "Setting affinity for IRQ $irq"
  echo 2 > /proc/irq/$irq/smp_affinity_list  # Example: Set to second core, adjust as needed
done

# Networking configuration for high performance
cat << EOF > /etc/network/interfaces.d/100G.conf
auto lo
iface lo inet loopback
```

```
# Configure interfaces for each NIC port
for i in {0..7}; do
  echo "auto eth$i" >> /etc/network/interfaces.d/100G.conf
  echo "iface eth$i inet manual" >> /etc/network/interfaces.d/100G.conf
  echo "  mtu 9000" >> /etc/network/interfaces.d/100G.conf
done

# Apply network configuration
systemctl restart networking

# Check link status of all interfaces
for i in {0..7}; do
  ethtool eth$i | grep -E "Link detected|Speed"
done

echo "Configuration completed. Please verify network performance."
```

Post-Script Notes:

- **Verify** the system's performance with tools like `iperf3` or specific MTL benchmarks.
- **Monitor** system logs and performance metrics for any bottlenecks or unexpected behavior.
- **Tune** further based on your application's specific needs, especially around interrupt handling and CPU core allocation.
- **Security**: Ensure your network setup is secure since this configuration focuses on performance.

This script should give you a solid foundation for high-throughput networking on your setup. Remember to test thoroughly in your environment.

bash

Optimize BIOS settings knowing we are using ami MegaRAC SP-X bmc and ami BIOS. Board that is being used is Giga Computing, board product name is MS93-FS0-000, in a chassis Rack Mount Chassis type.

14 web pages

Here are the BIOS settings you should consider optimizing for your Giga Computing MS93-FS0-000 board with AMI MegaRAC SP-X BMC and AMI BIOS to achieve peak performance, especially focusing on network throughput and system stability:

BIOS Settings for High Performance:

1. **CPU Configuration:**
   - **Hyper-Threading**: Enable - Improves multi-threaded performance.

- **Turbo Boost**: Enable - Allows CPUs to run above their base operating frequency if conditions allow.
- **C-States**: Disable - Reduces latency by preventing the CPU from entering low power states.
- **Power & Performance Policy**: Set to "Performance" - Optimizes for maximum performance over power savings.

2. **Memory Configuration:**
   - **Memory Speed**: Set to the maximum supported speed by your RAM modules.
   - **Memory Interleaving**: Enable - Improves memory bandwidth.
   - **NUMA (Non-Uniform Memory Access)**: Enable - Optimizes memory access for multi-socket systems like yours.
3. **PCIe/PCI Configuration:**
   - **PCIe ASPM (Active State Power Management)**: Disable - Ensures full bandwidth availability for PCIe devices like your network cards.
   - **PCIe Link Speed**: Set to maximum supported speed (e.g., Gen4 if available).
4. **Network Configuration:**
   - **SR-IOV (Single Root I/O Virtualization)**: Enable - Allows for better network performance by virtualizing the NICs.
   - **VT-d (Intel Virtualization Technology for Directed I/O)**: Enable - Enhances I/O virtualization performance.
5. **Boot Settings:**
   - **Boot Mode**: UEFI - More modern and offers better performance for newer hardware.
   - **Fast Boot**: Enable - Reduces boot time which can be beneficial for maintenance tasks.
6. **Power Management:**
   - **Power Supply Idle Control**: Disable - Keeps power supply in full performance mode.
   - **CPU P-state Control**: Set to "Performance" - Ensures CPUs run at full performance.
7. **Security Settings:**
   - Ensure secure boot and other security features are configured appropriately, but not at the expense of performance for your use case.
8. **Miscellaneous:**
   - **Wake On LAN**: Enable if needed for remote management; otherwise, disable to avoid unnecessary power usage.
   - **Above 4G Decoding**: Enable - Allows mapping of memory above 4GB, beneficial for larger memory systems.

BMC Specific Settings via MegaRAC SP-X:

- **Remote Access**: Ensure settings are optimized for minimal latency if remote management is crucial.
- **Sensor Monitoring**: Adjust thresholds to avoid unnecessary alerts during high load.

- **Firmware Update**: Regularly check for updates; ensure you're on the latest stable version of the BMC and BIOS firmware.

Additional Considerations:

- **Firmware**: Check for any BIOS or BMC firmware updates from AMI or Giga Computing that might include performance enhancements or bug fixes.
- **Documentation**: Refer to the specific documentation for your board from Giga Computing for any board-specific optimizations or settings.
- **Testing**: After adjusting these settings, thoroughly test your setup to ensure stability and performance gains.

These settings aim to maximize throughput for your network cards while ensuring the system runs at peak efficiency. Remember, these optimizations might increase power consumption and reduce the lifespan of hardware if not balanced with adequate cooling and maintenance.

Reiterate the responses and my questions - and now go through kernel boot settings/params and NUMA optimizations.

Your Questions:

1. **Initial Question**: You asked about optimizing system configuration for fully saturated throughput for 8x100G NIC cards under Ubuntu 22.04 using Intel Ice drivers and Media Transport Library (MTL), with a specific hardware setup including 2x Xeon Platinum CPUs, 128GiB RAM per CPU, and 4 Intel E810 NIC cards.
2. **Follow-up Question**: You asked for BIOS settings optimizations for an AMI MegaRAC SP-X BMC and AMI BIOS on a Giga Computing MS93-FS0-000 board in a rack mount chassis.

Responses:

Initial Response: Provided a detailed script for system configuration, network setup, driver installation, hugepages setup, and performance tweaks for the network interfaces.

Follow-up Response: Outlined BIOS settings for high performance, focusing on CPU, memory, PCIe, network, boot, power, and security configurations, tailored for your specific hardware and BMC setup.

Kernel Boot Settings/Parameters for Performance:

Here are some kernel boot parameters you might want to consider adding to your `GRUB_CMDLINE_LINUX_DEFAULT` in `/etc/default/grub`:

- **Huge Pages**:

```
default_hugepagesz=1G hugepagesz=1G hugepages=128
```

This ensures the system boots with huge pages of size 1GB, with 128 such pages initially allocated.

- **NUMA Balancing**:

  ```
  numa_balancing=disable
  ```

  Disables automatic NUMA balancing, assuming you'll manually manage memory allocation.

- **IOMMU (for VT-d)**:

  ```
  intel_iommu=on iommu=pt
  ```

  Enables IOMMU for better I/O performance and passthrough.

- **CPU Performance**:

  ```
  processor.max_cstate=1 intel_idle.max_cstate=0
  ```

  Disables deeper C-states for lower latency.

- **Network Performance**:

  ```
  msi=1 pcie_aspm=off
  ```

  Enabling MSI for better interrupt handling, and disabling ASPM for maximum PCIe bandwidth.

- **General Performance**:

  ```
  isolcpus=3-15,19-31 nohz_full=3-15,19-31 rcu_nocbs=3-15,19-31
  ```

  Isolate cores for workload, disable timer ticks on those cores, and move RCU callbacks off those cores for low-latency tasks.

After adding these parameters, update GRUB:

```bash
sudo update-grub
```

bash

NUMA Optimizations:

- **Memory Binding**: Use `numactl` to bind memory and processes to specific NUMA nodes. For instance:

```bash
numactl --membind=0,1 --cpunodebind=0,1 your_command
```

This binds memory to NUMA nodes 0 and 1 and processes to the CPUs in those nodes.

- **Taskset for CPU Affinity**: Manually set CPU affinity for critical processes to ensure they run on the same NUMA node as their memory:

```bash
taskset -c 0-15 your_process
```

- **Huge Pages Allocation**: Ensure that applications are using huge pages. Some applications might need specific flags or environment variables to utilize huge pages.
- **Interleave Memory**: For applications that span multiple NUMA nodes:

```bash
numactl --interleave=all your_command
```

This interleaves memory allocations across all NUMA nodes which can benefit applications with a large memory footprint.

- **Check NUMA Topology**: Use tools like `numactl --hardware` to understand your system's NUMA topology and adjust your strategy accordingly.
- **Avoid NUMA Cross-Traffic**: Try to keep processes and their data on the same NUMA node to minimize cross-node memory access which can significantly degrade performance.
- **Monitor Performance**: Use `perf` or `numastat` to monitor NUMA-related performance metrics, adjusting your setup based on real data.

These optimizations focus on enhancing your system's performance specifically for high-throughput networking scenarios while managing the complexities of multi-socket, NUMA architecture systems. Remember, these settings should be tested in your environment to ensure stability and performance.

Try once more - you acting like a pussy not a high tech guy. Reiterate and try once more! Remember that we are using lcores in the process and DPDK.

Reiteration:

Your Questions:

1. **Initial Question**: You're looking to achieve maximum throughput with 8x100G NIC cards on Ubuntu 22.04, using Intel Ice drivers, MTL, with hardware including 2x Xeon Platinum CPUs, 128GiB RAM per

CPU, and 4 Intel E810 NICs.

2. **Follow-up Question**: Asked for BIOS optimizations for AMI MegaRAC SP-X BMC and AMI BIOS on Giga Computing MS93-FS0-000 in a rack mount chassis.

Responses:

Initial Response: Detailed script for system and network configuration.

Follow-up Response: BIOS settings for performance.

Kernel Boot Settings/Parameters for High-Performance Networking with DPDK:

Update `/etc/default/grub` with these parameters:

- **Huge Pages**:

```
default_hugepagesz=1G hugepagesz=1G hugepages=128
```

  This configures 1GB huge pages for DPDK.

- **Disable CPU C-States & Turbo Boost**:

```
processor.max_cstate=1 intel_idle.max_cstate=0 intel_pstate=disable
```

  Essential for DPDK to avoid latency spikes.

- **IOMMU for VT-d**:

```
intel_iommu=on iommu=pt
```

- **PCIe Settings**:

```
pcie_aspm=off
```

  Ensures PCIe links maintain maximum performance.

- **Isolate Cores for DPDK**:

```
isolcpus=3-15,19-31 nohz_full=3-15,19-31 rcu_nocbs=3-15,19-31
```

  Isolates cores for DPDK lcores, disabling timer ticks and moving RCU callbacks.

```bash
sudo update-grub
```

bash

NUMA and DPDK Optimizations:

- **DPDK Configuration**: Use `dpdk-devbind.py` to bind your NICs to DPDK's UIO or VFIO driver:

  ```bash
  sudo dpdk-devbind.py --status
  sudo dpdk-devbind.py -b uio_pci_generic <PCI_BDF>
  ```

  bash

- **Lcore Mapping**: Configure lcores with NUMA awareness. Use `dpdk_lcore_mask` for setting the mask of lcores that DPDK can use. Here's how you might do it in a DPDK application:

  ```c
  EAL: lcore 0 is ready (tid=0x7f7c32f07700)
  EAL: lcore 1 is ready (tid=0x7f7c32ef6700)
  // Configure lcore mask to use only specific cores, ensuring they're on the same NUMA node as t
  ```

  c

- **NUMA-Aware Memory Allocation**:
  - In DPDK's `rte_eal_init()`, ensure you're using the right memory channel with `--socket-mem` specifying memory per NUMA node.
- **Binding Applications to NUMA Nodes**:
  - Use `numactl` for binding:

    ```bash
    numactl --membind=0 --cpunodebind=0 dpdk-testpmd
    ```

    bash

- **Huge Pages Setup**:
  - Ensure your application uses huge pages by setting `EAL` parameters:

    ```bash
    -n <number_of_channels> --socket-mem <memory_per_node>
    ```

    bash

- **Avoid NUMA Cross-Traffic in DPDK**:
  - Configure your application to allocate memory and use lcores from the same NUMA node to reduce latency.
- **Performance Monitoring**:

- Use DPDK's built-in tools or `perf` with DPDK to monitor performance, especially focusing on NUMA-related metrics.

Additional DPDK Tips:

- **SR-IOV**: Enable SR-IOV in BIOS and use `dpdk-devbind.py` to bind Virtual Functions (VFs) to DPDK drivers for better performance.
- **Polling Mode**: Use polling mode drivers in DPDK for lower latency and higher throughput.
- **Interrupt Coalescing**: Adjust interrupt coalescing settings for your NICs through DPDK configurations to balance between latency and CPU usage.
- **Power Management**: Keep power management settings in mind; you might want to disable or tune them for consistent performance.

This setup should give you a solid foundation for pushing your network throughput to its limits using DPDK and managing NUMA efficiently. Remember, real-world testing is key to validate these configurations in your specific environment.

DPDK performance tuning

Here are some key strategies for tuning DPDK performance, focusing on achieving maximum throughput and minimizing latency:

Hardware Configuration:

- **BIOS/UEFI Settings**:
  - **Disable C-States**: Reduces latency by ensuring CPUs are always in a high-performance state.
  - **Enable VT-d**: Enhances I/O virtualization performance.
  - **Enable SR-IOV**: For better network performance with virtual functions.
  - **Set Power Management to Performance**: Maximizes CPU performance over power efficiency.

Ask anything

DeepSearch Think Attach

Kernel and Boot Parameters:

- **GRUB Configuration**:
  - Add the following to `/etc/default/grub`'s `GRUB_CMDLINE_LINUX_DEFAULT`:

```
default_hugepagesz=1G hugepagesz=1G hugepages=128 isolcpus=3-15,19-31 nohz_full=3-15,19-31
```

  - Update GRUB after these changes.

DPDK Specific Tuning:

- **Huge Pages**:
  - Use 1GB huge pages for better TLB efficiency. Ensure your application uses these by configuring `--socket-mem` for DPDK's `rte_eal_init()`.
- **Lcore Configuration**:
  - Use `dpdk_lcore_mask` to specify which logical cores DPDK should use, ensuring these cores are isolated and on the same NUMA node as the data they'll primarily access.
- **Driver Binding**:
  - Use `dpdk-devbind.py` to bind NICs to DPDK's UIO or VFIO driver:

```bash
sudo dpdk-devbind.py --status
sudo dpdk-devbind.py -b uio_pci_generic <PCI_BDF>
```

bash

- **Interrupts and Polling**:
  - Configure for polling mode (`--no-interrupt`) in your DPDK application to reduce latency from interrupts.
  - Adjust interrupt coalescing settings if using interrupts for balancing between CPU load and latency.
- **Memory Pool Configuration**:
  - Tune the number of memory pools (`--mbufs`) based on your application's traffic patterns. More mbufs can help with bursty traffic but at the cost of memory.
- **RX/TX Queue Sizes**:
  - Increase `RX` and `TX` queue sizes for better handling of packet bursts, e.g., `--rxq=4096 --txq=4096`.
- **Flow Control and Offloading**:
  - Enable or disable hardware offload features like checksum offload, TSO (TCP Segmentation Offload), LRO (Large Receive Offload) based on your workload. Sometimes disabling offloads can provide better control over packet processing.
- **RSS (Receive Side Scaling)**:
  - Enable RSS to distribute incoming packets across multiple cores for better scalability, but be mindful of NUMA node placement.
- **NUMA Aware Configuration**:
  - Ensure that packet processing cores are on the same NUMA node as the NIC and memory they use. Use `--vdev` with NUMA-specific options when possible.

Application Level:

- **Batching**:

- Implement batching in your application for packet processing to reduce the CPU overhead per packet.

- **Optimized Memory Usage**:
  - Use direct memory access where possible and minimize cache misses by aligning data structures with cache line sizes.
- **Thread Affinity**:
  - Use `pthread_setaffinity_np()` or equivalent to pin application threads to specific cores for better cache utilization and reduced context switching.

Performance Monitoring and Tuning:

- **Use DPDK Tools**:
  - `dpdk-pdump`, `dpdk-proc-info`, and `dpdk-l2fwd` can provide insights into performance bottlenecks.
- **External Tools**:
  - Use `perf` for detailed CPU and cache performance metrics. Look for cache misses, TLB misses, etc.
- **Stress Test**:
  - Use tools like `pktgen-dpdk` to generate traffic and test your system under various conditions.
- **Iterative Tuning**:
  - Performance tuning is iterative. Measure, adjust, measure again. Look at metrics like packets per second, CPU utilization, and latency.

Remember, each system is unique, and what works best can vary based on your specific hardware, network topology, and application requirements. Continuous monitoring and tweaking based on real-world performance data are key to achieving peak performance with DPDK.

No go once more and reiterate on OS settings in form of script that you presented in the first place. Make everything as efficient as you can imagine.

Here's an enhanced script focusing on OS settings for maximum DPDK performance:

```bash
#!/bin/bash

# Update system packages
apt update && apt upgrade -y

# Install necessary packages
```

```
apt install -y build-essential dkms linux-headers-$(uname -r) ethtool pciutils numactl dpdk dpdk-de
```



```
# Hugepages setup
echo "vm.nr_hugepages=128" >> /etc/sysctl.conf
echo "vm.hugepagesz=1G" >> /etc/sysctl.conf
sysctl -p

# Create and mount hugetlbfs for each NUMA node
mkdir -p /mnt/huge_{0,1}
mount -t hugetlbfs nodev /mnt/huge_0 -o pagesize=1G
mount -t hugetlbfs nodev /mnt/huge_1 -o pagesize=1G
echo "nodev /mnt/huge_0 hugetlbfs pagesize=1G 0 0" >> /etc/fstab
echo "nodev /mnt/huge_1 hugetlbfs pagesize=1G 0 0" >> /etc/fstab

# DPDK driver binding
dpdk-devbind.py --status
for bdf in $(dpdk-devbind.py --status | grep 'If=vfio-pci' | awk '{print $1}'); do
    dpdk-devbind.py -b vfio-pci $bdf
done

# Kernel modules for DPDK
modprobe uio
modprobe uio_pci_generic
modprobe vfio-pci

# Disable irqbalance
systemctl stop irqbalance
systemctl disable irqbalance

# CPU Isolation
echo "GRUB_CMDLINE_LINUX_DEFAULT=\"isolcpus=3-15,19-31 nohz_full=3-15,19-31 rcu_nocbs=3-15,19-31 in
update-grub

# Ensure core isolation on boot
echo 3-15,19-31 > /sys/devices/system/cpu/isolated

# Network interface tuning for DPDK
for i in {0..7}; do
  interface="eth$i"
  ethtool -G $interface rx 4096 tx 4096
  ethtool -C $interface adaptive-rx off adaptive-tx off rx-usecs 100 tx-usecs 100
  ethtool -K $interface rx off tx off
done

# Disable unnecessary services for performance
systemctl disable NetworkManager
systemctl stop NetworkManager
systemctl mask NetworkManager

# Set up sysctl parameters for performance
cat << EOF >> /etc/sysctl.conf
net.core.rmem_max=26214400
net.core.wmem_max=26214400
net.core.rmem_default=26214400
net.core.wmem_default=26214400
net.ipv4.tcp_rmem=4096 87380 26214400
```

```
net.ipv4.tcp_wmem=4096 65536 26214400
net.ipv4.tcp_mem=26214400 26214400 26214400
net.ipv4.tcp_window_scaling=1
net.ipv4.tcp_timestamps=0
net.ipv4.route.flush=1
EOF
sysctl -p

# NUMA-aware memory binding for DPDK (example command, adjust per your app)
# numactl --membind=0 --cpunodebind=0 dpdk-testpmd -- -i

# Set CPU affinity for NIC interrupts to spread across isolated cores
for irq in $(cat /proc/interrupts | grep ice | awk '{print $1}' | sed 's/://'); do
  echo "Setting affinity for IRQ $irq"
  echo 3 > /proc/irq/$irq/smp_affinity_list  # Example: Set to an isolated core, adjust based on yo
done

echo "System configuration for DPDK performance completed. Now reboot to apply the changes."
```


Notes on the Script:

- **Huge Pages**: Configured to use 1GB pages for DPDK, with mounts for each NUMA node to minimize cross-NUMA access.
- **DPDK Driver Binding**: Binds network interfaces to VFIO for DPDK use.
- **CPU Isolation**: Isolates cores for DPDK lcores, ensuring they're not used by other system processes.
- **Network Tuning**: Configures network interfaces for high throughput, disabling unnecessary features for DPDK.
- **Sysctl Settings**: Optimizes TCP/IP stack for high-bandwidth, low-latency scenarios.
- **Service Management**: Disables NetworkManager to reduce system noise.
- **NUMA Awareness**: The script suggests using `numactl` for binding applications to specific NUMA nodes, although you'll need to do this per application.

Important: After running this script, reboot your system for changes to take effect. After rebooting, verify your setup with DPDK applications, adjusting lcore usage and memory configuration based on your application's needs and observed performance. Remember, this script makes assumptions about your hardware; you might need to adjust core numbers or other settings based on your specific system

Great work! Now what is left is NUMA node optimizations. Go the script approach once more!

Here's a script focused on NUMA node optimizations for your setup:

bash

Grok 3 beta

```bash
#!/bin/bash

# Ensure numactl is installed
if ! command -v numactl &> /dev/null; then
    apt install -y numactl
fi

# Get NUMA node information
numactl --hardware

# Define NUMA nodes based on your hardware (adjust these according to your system's actual NUMA con
NUMA_NODE_0=0
NUMA_NODE_1=1

# Function to bind process to a single NUMA node
bind_to_numa() {
    local node=$1
    local command=$2
    numactl --membind=$node --cpunodebind=$node $command
}

# Function to interleave memory across all NUMA nodes
interleave_memory() {
    local command=$1
    numactl --interleave=all $command
}

# Example: Starting DPDK applications with NUMA binding
# Adjust paths and commands as per your setup
# Binding to NUMA Node 0 for DPDK testpmd
bind_to_numa $NUMA_NODE_0 "dpdk-testpmd -- -i"

# Binding to NUMA Node 1 for another DPDK app (example)
bind_to_numa $NUMA_NODE_1 "dpdk-l2fwd -- -i"

# Example of interleaving memory for applications that might benefit from it
# This can be useful for applications with large memory footprints spanning multiple NUMA nodes
interleave_memory "your_large_memory_app"

# Optimize IRQ affinity
# Note: You need to adjust these based on your actual core topology and IRQ numbers
for irq in $(cat /proc/interrupts | grep ice | awk '{print $1}' | sed 's/://'); do
    # Spread IRQs across isolated cores on both NUMA nodes
    echo "Setting affinity for IRQ $irq"
    if [[ $(($irq % 2)) -eq 0 ]]; then
        echo 3 > /proc/irq/$irq/smp_affinity_list  # Example: Core on NUMA 0
    else
        echo 19 > /proc/irq/$irq/smp_affinity_list  # Example: Core on NUMA 1
    fi
done

# Check NUMA statistics for processes
# Example for checking NUMA stats of a running process (replace PID with actual process ID)
PID=$(pgrep dpdk-testpmd)
```

```
if [ -n "$PID" ]; then
    numastat -p $PID
else
    echo "dpdk-testpmd not running or PID not found."
fi

# Adjust memory allocation policy for the system (if not already set by previous setups)
echo "interleave" > /sys/kernel/mm/transparent_hugepage/enabled
echo "never" > /sys/kernel/mm/transparent_hugepage/defrag

echo "NUMA optimizations applied. Monitor your applications for performance improvements."
```

Notes:

- **NUMA Node Definition**: Adjust `NUMA_NODE_0` and `NUMA_NODE_1` based on your actual system's NUMA configuration as shown by `numactl --hardware`.
- **Binding**: The `bind_to_numa` function binds processes to a specific NUMA node for both memory and CPU. This reduces cross-NUMA traffic but requires you to know which part of your workload should be on which node.
- **Interleaving**: The `interleave_memory` function shows how to distribute memory allocation across all NUMA nodes, which can be beneficial for applications with large memory footprints.
- **IRQ Affinity**: Distributing IRQs across cores on different NUMA nodes helps in balancing load but requires knowledge of your core layout. The example script alternates IRQs between nodes.
- **NUMA Stats**: Checking `numastat` for processes can help you understand if your NUMA bindings are effective.
- **Transparent Huge Pages**: Disabling or setting to interleave can help in scenarios where you don't want the OS to automatically manage huge pages, which might interfere with your DPDK settings.

Remember, these optimizations should be tested with your specific workload to ensure they're beneficial. Also, you might need to run this script or parts of it with root privileges for full functionality.

CPU affinity optimization

Here's a script focusing on CPU affinity optimization, particularly for high-performance scenarios like DPDK:

```bash
#!/bin/bash

# Ensure taskset is installed
if ! command -v taskset &> /dev/null; then
    apt install -y util-linux
fi
```

```
# Function to set CPU affinity for a process
set_affinity() {
    local pid=$1
    local cores=$2
    taskset -pc $cores $pid
}

# Function to set CPU affinity at process start
start_with_affinity() {
    local cores=$1
    shift
    taskset -c $cores "$@"
}

# Example: Set affinity for existing DPDK processes
# Adjust according to your actual core isolation and DPDK application needs
DPDK_TESTPMD_PID=$(pgrep dpdk-testpmd)
if [ -n "$DPDK_TESTPMD_PID" ]; then
    set_affinity $DPDK_TESTPMD_PID 3-15
else
    echo "dpdk-testpmd not running or PID not found for setting affinity."
fi

# Example: Start a DPDK application with specific CPU affinity
# Note: This example assumes you've isolated cores 3-15 and 19-31
start_with_affinity 3-15 dpdk-testpmd -- -i

# Example: Set affinity for another DPDK app on different cores (NUMA node 1)
DPDK_L2FWD_PID=$(pgrep dpdk-l2fwd)
if [ -n "$DPDK_L2FWD_PID" ]; then
    set_affinity $DPDK_L2FWD_PID 19-31
else
    echo "dpdk-l2fwd not running or PID not found for setting affinity."
fi

# Optimize IRQ affinity for NICs
# This assumes you know which IRQs belong to NICs and which cores they should be pinned to
for irq in $(cat /proc/interrupts | grep ice | awk '{print $1}' | sed 's/://'); do
    # Example: Alternating between cores on different NUMA nodes for load balancing
    if [[ $(($irq % 2)) -eq 0 ]]; then
        echo 3 > /proc/irq/$irq/smp_affinity_list  # Core on NUMA node 0
    else
        echo 19 > /proc/irq/$irq/smp_affinity_list  # Core on NUMA node 1
    fi
done

# Example: Automatically set affinity for new processes (needs adjustment for your app)
# This monitors for new processes with specific names and sets their affinity
while true; do
    for proc_name in "dpdk-testpmd" "dpdk-l2fwd"; do
        NEW_PID=$(pgrep -n $proc_name)
        if [ -n "$NEW_PID" ]; then
            # Check if affinity was already set to avoid redundant operations
            current_affinity=$(taskset -p $NEW_PID | awk '{print $NF}')
            if [ "$current_affinity" != "3-15" ] && [ "$proc_name" == "dpdk-testpmd" ]; then
                set_affinity $NEW_PID 3-15
```

```
            elif [ "$current_affinity" != "19-31" ] && [ "$proc_name" == "dpdk-l2fwd" ]; then
                set_affinity $NEW_PID 19-31
            fi
        fi
    done
    sleep 10
done &

echo "CPU affinity optimizations applied. Monitor performance to ensure effectiveness."
```

Notes:

- **Affinity Setting**: The script uses `taskset` for setting CPU affinity, which is useful for fine-tuning where processes run on your CPU cores.
- **Existing Processes**: It checks if DPDK processes like `dpdk-testpmd` or `dpdk-l2fwd` are already running to set their affinity.
- **Starting Processes**: Demonstrates how to start processes with a predefined affinity.
- **IRQ Affinity**: Optimizes network interrupts to spread them across isolated cores, potentially on different NUMA nodes for better load distribution.
- **Automated Monitoring**: Includes a loop to automatically adjust the affinity of new instances of specified processes, which can be particularly useful in environments where processes are dynamically started.
- **Core Selection**: The core numbers (e.g., 3-15, 19-31) should align with your system's isolated cores from previous scripts. Adjust these based on your actual CPU topology and NUMA configuration.

Remember, CPU affinity settings are crucial for performance in high-throughput scenarios but require careful monitoring and adjustment based on your workload's behavior and the performance metrics you observe.

GPU affinity settings

Here's a script for managing GPU affinity settings, particularly focusing on binding processes to specific GPUs, which is less straightforward compared to CPU affinity due to the nature of GPU operations and system architecture. This script assumes you're working in a Linux environment with NVIDIA GPUs, where `nvidia-smi` and `CUDA` are available for managing GPU resources:

```bash
#!/bin/bash

# Ensure NVIDIA tools are installed
if ! command -v nvidia-smi &> /dev/null; then
    echo "NVIDIA drivers and CUDA toolkit not found. Please install them."
    exit 1
fi
```

```bash
# Function to get GPU ID from index
get_gpu_id() {
    local index=$1
    echo $(nvidia-smi --query-gpu=index --format=csv,noheader | sed -n "$((index+1))p")
}

# Function to set GPU affinity for a process
set_gpu_affinity() {
    local pid=$1
    local gpu_index=$2
    local gpu_id=$(get_gpu_id $gpu_index)

    # Set the CUDA_VISIBLE_DEVICES environment variable for the process
    # This approach effectively binds the process to use only the specified GPU
    echo "Setting GPU affinity for PID $pid to GPU index $gpu_index (ID $gpu_id)"
    CUDA_VISIBLE_DEVICES=$gpu_id taskset -pc 0-$(( $(nproc) - 1 )) $pid

    # Alternatively, for processes already running, you might need to use:
    # echo $gpu_id > /proc/$pid/environ
}

# Example: Set GPU affinity for an existing process
GPU_PID=$(pgrep -f "your_gpu_process")
if [ -n "$GPU_PID" ]; then
    set_gpu_affinity $GPU_PID 0 # Binding to GPU at index 0
else
    echo "GPU process not found for setting affinity."
fi

# Example: Start a GPU process with specific GPU affinity
CUDA_VISIBLE_DEVICES=1 your_gpu_application

# Monitor GPU usage to ensure correct binding
nvidia-smi -l 1

# Function to bind a new process to a specific GPU at startup
start_with_gpu_affinity() {
    local gpu_index=$1
    shift
    CUDA_VISIBLE_DEVICES=$(get_gpu_id $gpu_index) "$@"
}

# Example: Starting a new process with GPU 1 affinity
start_with_gpu_affinity 1 your_gpu_application

# Example: Setting affinity for multiple GPUs if needed
# This might be useful for applications that can leverage multiple GPUs
set_multi_gpu_affinity() {
    local pid=$1
    local gpu_indices="${@:2}"
    local gpu_ids=""
    for idx in $gpu_indices; do
        gpu_ids="${gpu_ids}$(get_gpu_id $idx),"
    done
    gpu_ids=${gpu_ids%,}
```

```
    echo "Setting GPU affinity for PID $pid to GPUs $gpu_indices (IDs $gpu_ids)"
    CUDA_VISIBLE_DEVICES=$gpu_ids taskset -pc 0-$(( $(nproc) - 1 )) $pid
}

# Example: Binding a process to multiple GPUs
MULTI_GPU_PID=$(pgrep -f "your_multi_gpu_process")
if [ -n "$MULTI_GPU_PID" ]; then
    set_multi_gpu_affinity $MULTI_GPU_PID 0 1 # Binding to GPU indices 0 and 1
else
    echo "Multi-GPU process not found for setting affinity."
fi

echo "GPU affinity settings applied. Monitor GPU usage with 'nvidia-smi' for verification."
```

Notes:

- **CUDA_VISIBLE_DEVICES**: This environment variable controls which GPUs are visible to CUDA applications. Setting it to a specific GPU ID (or list for multiple GPUs) effectively binds the application to use only those GPUs.
- **Taskset**: While mostly used for CPU affinity, it's included here to ensure the process uses all available CPU cores, which can be beneficial when the GPU process is CPU-bound.
- **GPU Index vs. ID**: `nvidia-smi` uses an index for display, but for programming, you typically use the ID. The script handles the conversion.
- **Monitoring**: Use `nvidia-smi` to monitor GPU usage to verify that your affinity settings are working as intended.
- **Multi-GPU**: For applications supporting multiple GPUs, you can specify multiple GPU indices in `CUDA_VISIBLE_DEVICES`.

Keep in mind, GPU affinity is more about ensuring exclusive access to hardware resources rather than optimizing performance in the same way CPU affinity does. The effectiveness can vary based on the application, its GPU utilization pattern, and the system's overall load.

OpenCL optimization

Optimizing OpenCL applications involves several layers of tuning, from device selection to kernel optimization. Here's a script with steps and examples to optimize OpenCL performance:

```bash
#!/bin/bash

# Ensure OpenCL is installed and configured
if ! command -v clinfo &> /dev/null; then
    echo "OpenCL tools (like clinfo for checking device info) are not installed. Install OpenCL dri
    exit 1
```

```
fi

# Function to check OpenCL devices
check_opencl_devices() {
    echo "Available OpenCL Devices:"
    clinfo | grep -A 3 'Device Name'
}

# Check available OpenCL devices
check_opencl_devices

# Device Selection Optimization
# Typically, you select devices in your OpenCL code or via environment variables
export OCL_PLATFORM=0  # Select platform index
export OCL_DEVICE=0    # Select device index within the platform

# Kernel Compilation Flags
# You can optimize kernel compilation with flags for better performance
export CLFLAGS="-cl-fast-relaxed-math -cl-mad-enable -cl-no-signed-zeros"

# Example of how you might compile an OpenCL kernel with optimization flags:
# This would be done in your C/C++ program, not in the shell:
# clBuildProgram(program, 1, &device_id, $CLFLAGS, NULL, NULL);

# Memory Management
# Ensure proper memory alignment and use of appropriate data types
echo "Ensure in your OpenCL code:
- Use __global for memory that will be accessed by the device
- Use __constant for read-only data not changing frequently
- Avoid unnecessary data transfers between host and device"

# Buffer Management
echo "Optimize buffer usage:
- Use clCreateBuffer with CL_MEM_READ_WRITE or CL_MEM_READ_ONLY as appropriate
- Use clEnqueueMapBuffer for better performance when dealing with large datasets"

# Work-group Size and Work-item Optimizations
echo "Adjust work-group sizes in your kernel:
- Experiment with different local_work_size to match your hardware's compute units
- Ensure global_work_size is divisible by local_work_size for full utilization"

# Vectorization
echo "Consider vector data types in kernels:
- Use float4, int4, etc., for SIMD-like operations where applicable"

# Profiling and Tuning
# Use OpenCL's built-in profiling to measure performance
echo "Enable profiling in your OpenCL queue:
- Use cl_queue_properties = CL_QUEUE_PROFILING_ENABLE in clCreateCommandQueue
- Use clGetEventProfilingInfo to analyze kernel execution times"

# Example: A simple script to run an OpenCL benchmark and gather profiling data
if [ -f "opencl_benchmark" ]; then
    ./opencl_benchmark -profile
else
    echo "You need an OpenCL benchmark application to run this profiling step."
```

```
fi

# Memory Coalescing
echo "Optimize for memory coalescing:
- Ensure data access patterns align with how GPUs fetch memory
- Group adjacent memory access in your kernel code"

# Avoid Branch Divergence
echo "Minimize branch divergence:
- Use if statements only when necessary, prefer conditional assignment"

# Asynchronous Operations
echo "Use asynchronous operations:
- Queue multiple operations with clEnqueueNDRangeKernel, allowing overlap of compute and data trans

# Device-specific Optimizations
# Check your device's capabilities and adjust accordingly (like max work-group size, local memory s
device_info=$(clinfo | grep -A 10 'Device Name')
echo "Device Specifics:
$device_info
Adjust your kernel parameters based on these specifications."

echo "Remember, OpenCL optimization often requires iterative testing and adjustment. Use the profil

# Note: The actual optimization would be in your OpenCL code or application settings, not in this s
```

Key Points for OpenCL Optimization:

- **Device Selection**: Choose the best device for your workload based on capabilities.
- **Compilation Flags**: Use appropriate flags to trade off between precision and performance.
- **Memory Management**: Optimize data transfers, use appropriate memory types, and ensure data alignment.
- **Work-group Configuration**: Tailor work-group sizes to the device's capabilities.
- **Vectorization**: Use vector data types for operations that can be parallelized.
- **Profiling**: Profile your application to identify bottlenecks.
- **Memory Access Patterns**: Optimize for coalesced memory access.
- **Reduce Branch Divergence**: Keep kernel execution paths uniform where possible.
- **Asynchronous Operations**: Utilize OpenCL's ability to queue operations for better overlap.

This script provides a framework for thinking about OpenCL optimizations rather than performing them directly, as most of these optimizations would happen within your OpenCL code or application setup.

Ok as we are having a good resolving conversation - i need a best known method for ... would be a script for drivers stack installation - i can only use and must have present on the system: a DPDK v23.11, custom patched and need to be build, ICE driver 1.14.9 and it is also custom patched version (patches from Media Transport Library GitHub repo in OpenVisualCloud). I need also the best/newest possible versions of iavf and irdma drivers.

15 web pages

Here's a script to install and configure the specific drivers you've mentioned, focusing on DPDK v23.11, ICE driver 1.14.9, and ensuring you have the latest versions of iavf and irdma:

```bash
#!/bin/bash

# Set up some variables for clarity
DPDK_VERSION="v23.11"
ICE_VERSION="1.14.9"
MTL_REPO="https://github.com/OpenVisualCloud/Media-Transport-Library"
DPDK_REPO="https://github.com/DPDK/dpdk"

# Ensure necessary tools are installed
sudo apt update && sudo apt install -y git make gcc build-essential linux-headers-$(uname -r) dkms

# DPDK installation
# Clone DPDK
git clone --branch $DPDK_VERSION $DPDK_REPO
cd dpdk || exit

# Apply custom patches (Assuming patches are in a directory named 'dpdk_patches')
if [ ! -d "../dpdk_patches" ]; then
    echo "No patch directory found. Please place DPDK patches in a directory named 'dpdk_patches' i
    exit 1
fi
for patch in ../dpdk_patches/*.patch; do
    [ -e "$patch" ] || continue
    git apply "$patch"
done

# Build DPDK
meson build
ninja -C build
sudo ninja -C build install

# Set DPDK env variables (for this session only, add to .bashrc or similar for persistence)
export RTE_SDK=$(pwd)
export RTE_TARGET=x86_64-native-linuxapp-gcc

# ICE driver installation
# Clone ICE driver
git clone --branch $ICE_VERSION --depth 1 https://github.com/intel/ethernet-linux-ice
cd ethernet-linux-ice || exit
```

```
# Apply ICE patches from MTL repo
if ! git clone --depth 1 $MTL_REPO; then
    echo "Failed to clone Media Transport Library repo."
    exit 1
fi
for patch in Media-Transport-Library/patches/ice/*.patch; do
    [ -e "$patch" ] || continue
    git apply "$patch"
done

# Build and install ICE driver
make
sudo make install
sudo modprobe ice

# iavf driver installation (latest from Intel's repo)
cd ..
git clone --depth 1 https://github.com/intel/ethernet-linux-iavf
cd ethernet-linux-iavf || exit
make
sudo make install
sudo modprobe iavf

# irdma driver installation (latest from Intel's repo)
cd ..
git clone --depth 1 https://github.com/intel/ethernet-linux-irdma
cd ethernet-linux-irdma || exit
make
sudo make install
sudo modprobe irdma

# Binding NICs to DPDK drivers
# WARNING: This will unbind your NICs from the kernel drivers. Ensure you're ready for this or have
sudo dpdk-devbind.py --status
echo "Manually bind your NICs to DPDK drivers using 'dpdk-devbind.py -b <driver> <PCI_ID>'"
# Example:
# sudo dpdk-devbind.py -b vfio-pci 0000:01:00.0

# Check installed modules
lsmod | grep -E "ice|iavf|irdma"

echo "Drivers installation completed. Please manually bind your hardware to the appropriate drivers
echo "Remember to add RTE_SDK and RTE_TARGET to your environment for DPDK applications."
```

Notes:

- **Patches**: This script assumes you have the necessary patches for DPDK and ICE in directories named `dpdk_patches` and that they're compatible with the versions you're using.
- **DPDK Environment**: The `RTE_SDK` and `RTE_TARGET` environment variables are set for this session. You'll need to add them to your environment settings (like `.bashrc`) for persistence.

- **Binding to DPDK**: The script doesn't automatically bind NICs to DPDK drivers due to the risk involved. You must do this manually, ensuring you're prepared for network disruption.
- **Kernel Modules**: Ensure your kernel supports these modules or compile them into your kernel if necessary.
- **Permissions**: Most commands require root or sudo access. Be careful when running this script to avoid system damage.
- **Latest Versions**: For iavf and irdma, we're pulling from the main branches, which should give you the latest stable versions at the time of cloning.

Remember to test your setup after installation to ensure everything functions as expected.