Technical white paper

Optimizing Linux performance on HP Integrity Superdome X

Table of contents

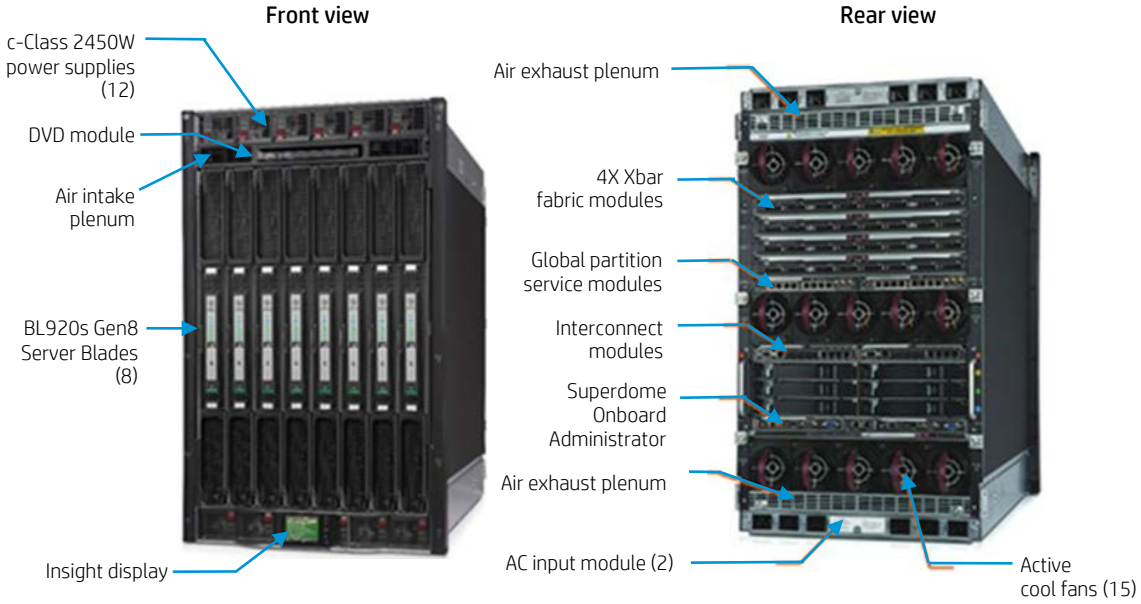
Introduction

The HP Integrity Superdome X Server blends trusted Integrity Superdome reliability with a standard x86 architecture. Building on the mission-critical capabilities of Integrity Superdome 2, Superdome X offers breakthrough scalability and efficiency to optimize your resource utilization and licensing costs. This document describes the Superdome X system and offers recommendations for achieving the highest performance when used in a Linux® environment.

Hardware overview

Superdome X is comprised of an enclosure and one or more two-socket BL920s Gen8 blades. The system can contain a maximum of 8 blades for a total of 16 processor sockets. Hard partitioning capabilities (nPar) with electrical isolation provides flexible configurations that can be tailored to a customer's needs.

Figure 1. HP Integrity Superdome X system.



Each BL920s Gen8 blade feature:

- Two Intel® Xeon® E7 v2 processor sockets
- 48 DDR3 DIMM slots accommodating:
  - 32 GB PC3-14900 DDR3 ECC registered Load Reduced DIMMS
  - 16 GB PC3-12800 DDR3 ECC registered DIMMS
- Two FlexLOM slots accommodating:
  - HP Ethernet 10Gbe 2-port 560FLB adapters
  - HP FlexFabric 10Gbe 2-port 534FLB Adapter
- Three mezzanine PCIe Gen3 slots accommodating:
  - HP Ethernet 10Gbe 2-port 560M adapters
  - HP FlexFabric 10Gbe 2-port 534M adapters
  - QMH2672 16Gb dual-port FC HBA

Each enclosure consists of up to eight BL920s Gen8 Server Blades, one upper midplane that accommodates four sx3000 Xbar Fabric modules, and one lower midplane that interfaces to I/O interconnect modules that plug into bays in the rear of the enclosure. The sx3000 provides break-through innovations including a fault-tolerant crossbar fabric, an error analysis engine, and hard partitioning capabilities, allowing Superdome X to set the standard for mission-critical x86 computing.

Additional details such as system architecture and blade partitioning can be found in the HP Integrity Superdome X System Architecture and RAS whitepaper and the HP Integrity Superdome X QuickSpecs.

Software overview

The Superdome X Operating System Support Matrix includes the following Linux distributions:

- Red Hat® Enterprise Linux (RHEL) 6.5 with maintenance update kernel 6.5.z kernel-2.6.32-431.20.3.el6 (or any later version). RHEL 6.6 and RHEL 7.0.
- SUSE Linux Enterprise Server (SLES) 11 SP3 with ProLiant Gen8 BigSMP Bootable Driver Kit which includes the required kernel: kernel-bigsmp-3.0.101-0.30.1.x86_64 (or later). Information about the Bootable Driver Kit is available at drivers.suse.com/hp/HP-ProLiant-Gen8-BigSMP/1.0/sle-11-sp3-x86_64/install-readme.html.

For more information on HP's Certified and Supported HP Servers for OS and Virtualization Software and latest listing of software drivers available for your server, please visit our Support Matrix at: hp.com/go/ossupport and our HP Integrity Superdome X support page.

Non-Uniform Memory Access considerations

Modern processors like the Intel Xeon E7 v2 have integrated memory controllers, and thus have excellent access latencies to their local memory. Remote memory (or memory connected to a memory controller on another processor) have higher access latencies. This architecture is called Non-Uniform Memory Access or NUMA. Best performance is achieved when tasks and their associated memory objects are close together, preferably in the same socket or NUMA node. The Linux operating system includes algorithms that attempt to keep memory objects close to the CPU that accesses them. However, an application's tasks can migrate over time to CPUs in other NUMA nodes and away from their memory objects, resulting in reduced performance.

Task and memory object placement (numactl/taskset/cgroups)

The best method to constrain task migrations is to use the numactl(8) command to control the placement of memory objects and task threads.

```
[root@sdx ~]# numactl -m 1 -N 1 numactl –show
policy: bind
preferred node: 1
physcpubind: 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
cpubind: 1
nodebind: 1
membind: 1
```

The `numactl` command uses the NUMA policy library (libnuma) interface for memory allocation policy retrieval and control. Refer to the man pages numactl(8) and numa(3) for more information. The taskset(1) command controls placement of task threads and uses the default memory allocation policy which attempts to allocate memory from where the task is executing. The cpuset(7) command, a subset of control groups (cgroups), provides the ability to bind tasks to particular CPUs and their associated memory objects to particular memory nodes. Unlike numactl which only affects tasks launched via the command, existing tasks can be moved into and out of cpusets allowing much greater flexibility. Refer to the RHEL 6 Resource Management Guide for more information on cpusets and control groups.

Determining Task Memory Allocation Location

The /proc/<pid>/numa_maps file identifies a task's memory allocation for various objects and their location. This file displays each memory object for a particular task. The following example shows the entries for a current shell's heap and stack:

```
[root@sdx ~]# grep –e heap –e stack /proc/$$/numa_maps
0245f000 default heap anon=65 dirty=65 active=60 N1=65
7fff23318000 default stack anon=7 dirty=7 N1=7
```

In the second and third lines above, the first field is the start of the Virtual Memory Address (VMA) range. The second field is the memory allocation policy. The third field is the path to the mapped file, shared memory segment, or type. The `anon=` and `dirty=` show the number of pages. The `N<node>=` shows the number of pages allocated from each <node>.

Automatic NUMA Balancing

RHEL 7 includes automatic NUMA balancing functionality first introduced with 3.8 kernel. This feature attempts to improve performance by either moving tasks closer to the memory they are accessing or moving the memory pages closer to where the task is executing. These algorithms include:

- Periodic unmapping of process memory pages a little at a time. These unmapped pages result in NUMA-hinting faults allowing the kernel to track and process memory location.
- Migrate-on-Fault (MoF): This function moves memory pages to where the task accessing them is executing.
- task_numa_placement(): This kernel routine which moves running tasks closer to their memory objects.

While the Automatic NUMA Balancing feature benefits many workloads, some applications may experience increased page fault latency when pages are migrated (Oracle® completely disables Automatic NUMA Balancing for kernel builds for Oracle Linux). Automatic NUMA Balancing can be disabled by setting numa_balancing to 0 as follows:

```
[root@sdx ~]# sysctl -w kernel.numa_balancing=0
```

or

```
[root@sdx ~]# echo 0 > /proc/sys/kernel/numa_balancing
```

Note
Disabling Automatic NUMA Balancing may require you to explicitly set the configuration for starting processes to manually balance the workload among the available NUMA nodes and avoid overuse of processor and memory resources.

Power consumption versus performance

Power savings and high performance in modern servers are usually thought to be mutually exclusive. Generally, lower power consumption means less heat generated, and consequently lower performance levels. However, for large NUMA servers such as Superdome X, some workloads offer the opportunity to lower power usage and still maintain or improve performance.

Power management features

Superdome X has hardware and firmware features that can reduce power consumed by the CPU and peripheral circuitry when periods of idle occur. This section discusses tuning those features that have a direct impact on application performance.

C-states

C-states are HW sleep states that power down different portions of the CPU peripheral circuitry and have multiple levels—each corresponding to greater power savings but also incurring longer latencies to wake up. While most C-states generally lower performance, there are some scenarios where allowing C-state 1 transitions on idle cores can improve performance on non-idle cores within the same processor by encouraging/enabling higher TurboBoost clock speeds. Table 1 describes common C-state behaviors, which can vary by processor type. The Intel Xeon E7-2890 V2 processors used in the Superdome X support C-states 0, 1, 1E, 3, and 6 (non-grayed in Table 2).

Table 1. C-states.

C state	Name	Description
C0	Operating state	CPU is fully turned on and executing instructions. It can be in one of P-states (P0 - Pn) which defines operational voltage and frequency.
C1	Halt	CPU main internal clocks are stopped. Bus interface unit and APIC are kept running at full speed.
C1E	Enhanced Halt	CPU main internal clocks are stopped and the CPU voltage is reduced. Bus interface unit and APIC are kept running. The frequency can be also reduced.
C2	Stop Clock	CPU internal and external clocks are stopped via hardware.
C3	Deep Sleep	CPU internal and external clocks are stopped, L1/L2 cache can be flushed.
C4	Deeper Sleep	CPU voltage is reduced.
C5	Enhanced Deeper Sleep	CPU voltage is reduced even more and the memory cache is turned off.
C6	Deep Power Down	Core states are saved into memory with low power consumption. It can reduce the CPU internal voltage to any value, including 0 V.
C7	Deeper Power Down	Same as C6 + flush of L3 cache.

Note that the Linux kernel defines a number of logical "C-states" 0 through 4, and will map the various CPU HW defined C-states above. To determine which HW C-state is used for the Linux OS-defined state use the following command:

```
# cat /sys/devices/system/cpu/cpu0/cpuidle/state<C-state number>/name
```

For example, the OS-defined meaning of C-state4 on Superdome X is the Ivy Bridge C6 state:

```
# cat /sys/devices/system/cpu/cpu0/cpuidle/state4/name
  C6-IVB
```

Recommendation
Allow only C-state 1 transitions to encourage Turboboost clock speed increases. To allow the intel_idle driver to use C-state1, add the following to the boot command line: " `intel_idle.max_cstate=1` ". Alternatively, use the userspace utilites `tuned-adm` profile "latency-performance", or with the `cpupower idle-set` command.

P-states

P-states control the CPU clock frequency within a range of settings supported by the CPU. The adjustments of speed are usually load-related and are not real-time, and are slow to adapt. In terms of performance impact, the lower CPU clock frequencies are clearly a performance degradation. For top performance we recommend the CPU clocks should be kept at full speed.

Recommendation
CPU clock should be kept at full speed by using the tuned-adm profile "latency-performance" or with the command `cpupower frequency-set –g performance.`

Utilities for managing power usage

C-States are managed at the kernel level by an idle driver—typically the intel_idle driver on RHEL7 or the acpi_idle driver on earlier releases. If the intel_idle driver is disabled, the acpi_idle driver will be used.

The following command lines can be used to determine which driver is being used:

```
# dmesg | grep acpi_idle
ACPI: acpi_idle yielding to intel_idle
```

...or

```
# cpupower idle-info ( more about this tool shortly)
CPUidle driver: intel_idle
```

To limit the allowed intel_idle driver C-state transition to C-state1 (the recommended setting) add the following parameter to the kernel boot command line:

```
intel_idle.max_cstate=1
```

...or if the acpi_idle driver is used, add:

```
processor.max_cstate=1
```

The intel_idle driver will always allow C-state1 transitions. This is the minimum value allowed for 'max_cstate'. Setting intel_idle.max_cstate to '0' actually disables the intel_idle driver completely, and will result in the acpi_idle driver being used. . If you want to eliminate all C-state transitions, you must disable the intel_idle driver and use the acpi_idle driver and the processor.max_cstate=0 option on the boot command line.

Userspace utilities can also control the C-states used by the idle drivers, making any bootline modifications unnecessary. Some make use of the Process Management Quality of Service (PM QoS) device file `/dev/cpu_dma_latency`. Processes can open this device file and write a minimum requested latency value to it and the kernel will attempt to limit C-state transitions to meet the requested value as long as the process keeps the file open. Another userspace technique to control C-states is to use the sysfs file system to enable or disable specific C-states on a per-CPU basis by editing the `/sys/devices/system/cpu/cpu<n>/cpuidle/state<n>/disable` files. In order to determine which methods are being used to control the C-states you will need to evaluate both the processes with the /dev/cpu_dma_latency file open (using the lsof utility), and the values of the sysfs 'disable' files mentioned above.

P-states are controlled by the kernel P-state driver. On RHEL7 for Superdome X, this is the intel_pstate driver. On earlier releases the acpi-cpufreq driver is used. The P-state driver will offer various speed governors such as 'on-demand', 'performance', 'powersave' and others.

cpupower

A common utility to control both C-states and P-states is cpupower, which is included with the cpupowerutils package. The CPU frequency changes are monitored/controlled using the cpupower frequency-info and cpupower frequency-set commands.

To show the CPU frequency/speed related parameters such as the P-state driver in use, the current governor, the current CPU speed, etc. use:

```
# cpupower frequency-info
```

To set the CPU speed governor for maximum performance (maintaining full clock speed) use:

```
# cpupower frequency-set -g performance
```

To monitor/control the C-states being controlled via the sysfs infrastructure, display the idle driver in use, as well as the HW C-state specific details, use the cpupower idle-info and cpupower idle-set commands.

To display the C-state settings use:

```
cpupower idle-info
```

To disable (enable) specific C-states use:

```
cpupower idle-set –d (-e) <C-state>
```

To allow only C-state1 transitions you would execute:

```
cpupower idle-set –d 4
cpupower idle-set –d 3
cpupower idle-set –d 2
```

tuned-adm

Tuned is a common RedHat-specific utility also used to control C-states and P-states. Tuned defines a number of settings that it groups together in different 'profiles'. The tunable parameters set in these profiles varies from RHEL6 to RHEL7, and include more than just the C-state and P-state settings. Tuned profiles can control the I/O elevator, use of Transparent Hugepages (THP), scheduler parameters, and more. For best performance on the Superdome X, we recommend using the **latency-performance** profile for RHEL6 and the **network-latency** profile on RHEL7. These profiles both limit C-state transitions to 1, and keep the CPU at maximum clock speed.

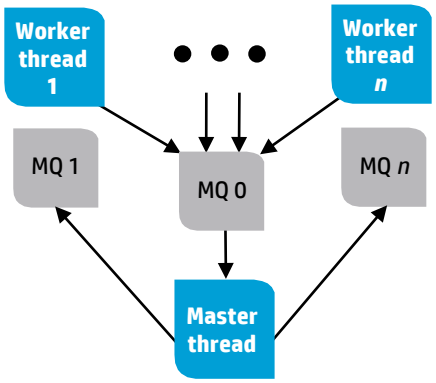
For more information on controlling power features on SLES11 refer to the following link: suse.com/documentation/sles11/singlehtml/book_sle_tuning/book_sle_tuning.html#cha.tuning.power.

For more information on RHEL7 tuned profile definitions and general configuration refer to the following webpage: access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Power_Management_Guide/Tuned.html.

Test environment

To illustrate how to configure the Superdome X server power management components for the best throughput and lowest latencies, we will use a latency sensitive message passing micro-benchmark. The turbostat utility is used to monitor the power usage statistics, since it also provides a good processor level summary of the most relevant HW features (clock speeds and C-states) used to control power usage. The message passing micro-benchmark contains a master thread that receives all messages on a single message queue (Figure 2). Worker threads send requests to the single master thread message queue (MQ 0) and await a reply on their own individual message queue (MQ 1 – MQ n). The test includes a designed bottleneck; a master thread's single message queue with synchronization locks being needed by all worker threads. The master thread is also CPU-bound and never blocks except to await a new message from a worker thread.

Figure 2. Test environment.



The test environment is designed to be latency-sensitive and demonstrate the effect of tuning efforts that improve latency through C-state and P-state changes. It also demonstrates the impact of better data locality through workload placement using CPU and node binding techniques. Proper workload placement can improve data locality and CPU efficiency while also freeing up nodes and CPUs to be available (unused) for deeper C-state and P-state power saving levels.

Test scenarios

The benchmark was executed in two scenarios. The first scenario used eight instances of the workload running in parallel. Each instance consisted of one master thread and 32 worker threads. The number of messages/sec was used as the performance metric while monitoring processor power usage at the same time. No CPU or node binding of processes was used, and both low-performance and high performance tuning values were tested.

The second test scenario used a single instance with 120 worker threads, with and without node binding. Both low-performance and high-performance tuning values were tested. The point of the second test scenario is to highlight the advantage of optimal workload placement and how it can improve performance while still allowing lower power usage levels. The low and high performance tuned values for both scenarios are shown in Table 2.

Table 2. Power consumption versus performance test configurations (both scenarios).

Tunable	Low performance configuration	High performance configuration
tuned-adm profile	balanced	latency-performance
cpupower frequency-set –g	powersave	N/A
Allowed C-states	all	0, 1

Test results

The first test scenario's results (Table 3) indicate the TPS improves by 30% and power usage increases by 6% while the workload is running. In the idle state however, the base power usage increases 317% from 182W up to 578W.

Table 3. Power consumption versus performance test results—first scenario.

Measurement	Low performance configuration	High performance configuration
Base power floor (all Processors) while idle	182 W	578 W
Total TPS, all 8 instances	867725 TPS	1131823 TPS
Lowest TPS by an instance	103714 TPS	115270 TPS
Highest TPS by an instance	117993 TPS	152808 TPS
Total power usage (all processors)	630 W	669 W
%C1 / %C3 / %C6	67.3 / 4.5 / 18.8	99.97 / 0 / 0
Avg CPU clock speed	1.73 GHz	3.19 GHz

The second test scenario's results (Table 5) indicate that the best TPS is, again, achieved using the high performance configuration tunings and the use of node binding. However, in the low-performance configuration set, if you use node binding the TPS is only 5% lower but at less than 50% the power usage.

Note
The use of node binding improves performance of the low performance configuration above that of the unbound high performance configuration (bolded values in Table 4). This underscores the importance and advantages of NUMA awareness within an application

Table 4. Power consumption versus performance test results—second scenario.

Measurement	Low performance configuration	High performance configuration
Base power floor (all Processors) while idle	182 W	578 W
TPS, no binding	91650 TPS	**306560 TPS**
TPS w/node 0 binding	**708430 TPS**	742929 TPS
Total power usage, no binding	311 W	598 W
Total power usage, node 0 binding	258 W	606 W

Optimizing network performance

A system's networking performance is typically defined by its scalability, bandwidth, and latency. These characteristics are determined by:

- External network topology
- Base speed of the underlying link technology
- Proper driver-level configuration
- Proper transport-layer configuration
- Optimal assignment of the CPU processing workload relative to the tasks/processes using the data
- Proper architecture and use of network services and infrastructure at the application level

Note
This paper focuses on components internal to the server but below the application design level.

Test environment

Our test environment uses the Netperf TCP_STREAM and UDP_RR tests to illustrate how to configure the Superdome X server for the best networking throughput and lowest latencies. The Netperf TCP_STREAM test produces a bulk data transfer stream while the UDP_RR test simulates latency sensitive request-reply transaction workloads. The UDP_RR test is similar to the latency-sensitive, request-response traffic profile common in Oracle RAC Global Cache Fusion traffic.

You can obtain the Netperf Utility and Documentation at netperf.org/netperf.

This exercise uses an 8-socket (120 core) Superdome X server using Intel 82599 10Gb FlexLOM NICs to connect to another Superdome X partition through the internal Intelligent Resilient Framework (IRF) path feature of the 6125XLG switch modules. Tests were also run between the Superdome X and DL360s Gen9 Servers accessed through an external 8212ZL 10Gb switch. Both single NIC performance and a 4-port NIC bond are evaluated.

The Superdome X 10Gb LOM NICs use the Intel ixgbe driver. This driver provides a number of CPU offload features enabled by default, as well as excellent receive packet steering (RPS) and receive flow steering (RFS) management. For a full description of the ixgbe driver features, see the README file included with the driver distribution at downloadmirror.intel.com/22919/eng/README.txt.

The ixgbe RPS management makes use of multiple receive queues (64 with the current driver), each with an interrupt assigned to a CPU in the same NUMA node as the NIC. A hash algorithm is used to assign packets to a queue. These queues dictate which CPUs will do the lower lever inbound receive processing before handing off the remainder of the inbound processing to a CPU closer to the where the owning task last ran (RFS). The close association of inbound processing on the CPU closest to the task improves data locality and thus CPU efficiency as the packet data is consumed. The ixgbe RFS implementation is referred to as 'Flow Director' in the ixgbe driver documentation. Only parameters that differ from default are shown. The test uses ixgbe driver revision 3.19.1-k.

Network traffic pattern simulation

The Netperf utility is used to generate both TCP stream traffic and UDP request-response traffic. This utility avoids, as much as possible, any interaction with non-network kernel components and resources such as file system I/O, extensive memory allocation, and dependencies on other services. Use of network applications (such as FTP, SCP, and NFS) for performance testing should be used only with a full understanding of their internal architecture, limitations, and their required resources.

The Netperf utility has a partner process called netserver (running on the remote system) that is used to accept and respond to the requested test commands. All tests used CPU binding for netserver tasks with a separate netserver for each of the 240 ports used in testing. Typical netserver syntax is:

```
/bin/taskset -c 0  netserver -p 30000
```

Some tests include multiple Netperf test loads running in parallel to demonstrate scalability characteristics.

Test configurations

The Netperf TCP_STREAM tests were conducted in two configurations. The first configuration is a Low Performance configuration where throughput results and latencies were expected to be non-optimal. Many of the tunable values in the Low Performance configuration are default values or former release default values. The second configuration is the High Performance configuration, with tunable values optimized for this specific test and configuration. Tunables with the largest impact on performance are in bolded text in Table 5.

Table 5. TCP_STREAM test configurations.

Tunable	Low-performance configuration (defaults)	High-performance configuration
Frame MTU size	1500	9000
ixgbe UDP HW RSS mode	disabled	SDFN
Bonded NICs	disabled	4-port bond, mode 4
Bond xmit_hash_policy	N/A	1
ixgbe rx_usecs	1	0
net.core.rmem_max	256K	16777216
net.core.wmem_max	256K	16777216
net.ipv4.tcp_rmem	4096 87380 4194304	4096 87380 16777216
net.ipv4.tcp_wmem	4096 16384 4194304	4096 16384 16777216
tuned-adm profile	balanced	latency-performance
cpupower frequency-set –g	powersave	N/A

TCP_STREAM test results

Using the same Netperf TCP_STREAM tests on the same test environment, the High Performance configuration resulted in a 21% improvement for the single stream test (Table 6). The single stream test can use nearly all available 10Gb NIC bandwidth.

Table 6. TCP_STREAM test results.

Measurement	Low-performance configuration (with 4 NIC bond)	High-performance configuration
Single TCP_STREAM	7.98 Gb/s	9.89 Gb/s
Multiple (15) TCP_STREAM	9.47 (29.8) Gb/s	36.4 Gb/s
Multiple (30) TCP_STREAM	9.45 (35.9) Gb/s	39.63 Gb/s
Multiple (45) TCP_STREAM	9.45 (37.9) Gb/s	39.99 Gb/s
Multiple (60) TCP_STREAM	9.66 (37.9) Gb/s	39.99 Gb/s

The Netperf TCP_STREAM test used the following typical options/settings:

```
/opt/netperf/netperf -p 30020 -H10.10.220.20 -fg -l30 -t TCP_STREAM -- -m16K -M16K -s4M -
S4M -D
```

...where:

-p = target TCP port number
-H = target host IP address
-f = Set output units
-l = test length in seconds
-- = test specific options follow
-m/M = send/recv message size
-s/S = local/remote socket buffer size
-D = set TCP_NODELAY socket option

Test utilities such as Netperf typically make explicit requests to set various socket parameters such as the send/recv socket buffer size (`-m/M` option). If a networking application does not make an explicit request it will receive the system default values using the sysctl parameters. The defaults are usually sufficient for good performance. However, at times it may be necessary to alter the system defaults.

Note
A legacy socket application ported to Linux may still request socket buffer sizes that are not optimal on Linux. An example would be a setsockopt( ) system call to request 128K socket buffer sizes, which is typical and usually adequate on other platforms. On Linux, this will reduce the TCP_STREAM throughput results by 20-30%. The default send/receive socket buffer sizes of 1-4MB are recommended.

Use of bonded NICs

Using bonded NICs has a side effect of randomizing the relative locality of the tasks and the NICs they are using. Any one connection will only use one NIC as determined by the xmit_hash_policy of the bond configuration. This means that some connections may be ideally aligned with the NIC (executing on the same NUMA node as the NIC hard interrupt assignment), while others might be assigned NICs in a remote NUMA node. The ideal vs. poor locality alignment can impact TCP_STREAM and UDP_RR results by as much as 20%. With many test streams running, the assignments average out and the overall NIC(s) speed becomes the limiting factor as a whole. Consider the results of the 15 TCP_STREAM test vs. the 30 TCP_STREAM test. The four 10Gb NICs in the trunk become the limiting factor and throughput tops out at ~40Gb/s as expected, regardless of the number of additional TCP_STREAM tasks running.

Another factor in the throughput results is the Linux kernel scheduler. Without explicit CPU or node binding, the tasks will tend to migrate to just about all CPUs on the system. This makes their locality with the NICs drift in and out of ideal alignment resulting in variations in performance. In other words, “your mileage may vary.”

UDP_RR Request-Response test results

Optimizing a request-response workload usually requires tuning for minimal network latency. The Netperf UDP_RR test workload involves minimal non-network overhead and is very latency sensitive. The request and response sizes were chosen to be similar to what you might see in Oracle RAC Global Cache Fusion traffic—small 300 byte requests and large 8K blocks returned.

Using the Netperf UDP_RR tests on the same test environment, the High Performance configuration resulted in an 86% improvement for the single UDP_RR test, and 37% – 65% improvement in the multiple UDP_RR test results (Table 7).

Table 7. UDP_RR Request-Response test results.

Measurement	Low-performance configuration	High-performance configuration
Single UDP_RR	7610 TPS, 0.50 Gb/s	14191 TPS, 0.93 Gb/s
Multiple (15) UDP_RR	118821 TPS, 7.8 Gb/s	195909 TPS, 12.8 Gb/s
Multiple (30) UDP_RR	225392 TPS, 14.8 Gb/s	361672 TPS, 23.7 Gb/s
Multiple (45) UDP_RR	333067 TPS, 21.8 Gb/s	445585 TPS, 29.2 Gb/s
Multiple (60) UDP_RR	402385 TPS, 26.4 Gb/s	539083 TPS, 35.4 Gb/s
Multiple (120) UDP_RR	575203 TPS, 37.7 Gb/s	600945 TPS, 39.4 Gb/s
Multiple (180) UDP_RR	584609 TPS, 38.3 Gb/s	606683 TPS, 39.7 Gb/s

The Netperf UDP_RR test used the following typical options/settings:

```
./netperf -p 30000 -H10.10.220.11 –l30 -t UDP_RR -- -r 300,8000 -s1M -S1M
```

…where:

- `-p` = target TCP port number
- `-H` = target host IP address
- `-l` = test length in seconds
- `--` = test specific options follow
- `-r` = send/recv message size
- `-s/S` = local/remote socket buffer size

Explanation of tunable settings used for testing

9000 byte MTU (Jumbo Frames)

The default Ethernet frame size is 1500 bytes, typically consisting of 1460 bytes of data payload and 40 bytes of UDP/TCP/IP header information. Using Jumbo frames can improve the efficiency of packet handling. For UDP protocol, any message larger than 1460 bytes would be fragmented into multiple IP datagrams and require reassembly at the remote end when received. Many driver algorithms used to steer traffic cannot determine optimal inbound queue assignment of these IP fragments and result in default, non-optimal processing of the frames. By using a 9000 byte MTU size, larger messages can be managed more efficiently. Using Jumbo frames does require configuration of the attached network switch equipment to properly support larger MTU sizes.

To set the frame MTU size:

ifconfig <interface> mtu 9000

ixgbe UDP HW RSS mode

By default, the ixgbe driver Receive Side Scaling mechanism (RSS) steers incoming packets toward its multiple queues based on its Flow Director logic. This default steering logic looks at the 4-tuple of Source IP, Destination IP, Source port, and Destination port numbers to decide which of its multiple queues to assign the incoming packets to. For UDP however, it only evaluates the Source and Destination IP addresses. This can result in poor distribution and overloading one queue for certain environments. The following ethtool command enables the use of 4-tuple steering logic for UDP packets resulting in more efficient and even UDP traffic processing:

```
ethtool -N  <interface>   rx-flow-hash   udp4   sdfn
```

For more information refer to downloadmirror.intel.com/22919/eng/README.txt.

Bonded NICs & Bond xmit_hash_policy

The use of multiple NICs in a bond (also referred to as a 'trunk') allows for greater aggregate bandwidth and greater sharing of CPU resources needed to support NICs and NIC drivers. Many bond configurations also include redundancy schemes. Connections are spread across the available NICs based on various policies. The 'xmit_hash_policy' chosen for the bond in this testing used the Level 3 and Level 4 protocol header information (IP & UDP/TCP port numbers), to hash to an assigned NIC. By default, interfaces in Linux are not bonded together.

For more details on bond usage and configurations, refer to access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Networking_Guide/sec-Using_Channel_Bonding.html.

ixgbe driver rx-usecs

The rx-usecs parameter controls the driver interrupt throttling mechanism. By default the driver uses a dynamic 'delay' in asserting interrupts in hopes that more packets can be processed within a single interrupt event, thus improving CPU efficiency. This added delay can impact latency-sensitive applications. Setting this value to zero disables the interrupt behavior. Disabling the driver interrupt throttling is done with the following command:

```
ethtool -C <interface>  rx-usecs  0
```

Refer to the "InterruptThottleRate" parameter discussed here downloadmirror.intel.com/22919/eng/README.txt for more details.

net.core.rmem_max

This sysctl tunable dictates the largest allowable receive socket buffer size. Defaults vary per OS version but we recommend 16MB.

net.core.wmem_max

This sysctl tunable dictates the largest allowed send socket buffer size. Defaults vary per OS version but we recommend 16MB.

net.ipv4.tcp_rmem

This sysctl tunable defines the default receive socket buffer size if no other size is explicitly requested. Defaults vary per OS version but we recommend 16MB.

net.ipv4.tcp_wmem

This sysctl tunable defines the default send socket buffer size if no other size is explicitly requested. Defaults vary per OS version but we recommend 16MB.

Power Savings (tuned-adm profile & cpupower speed governor settings)

To reduce I/O latencies, be sure to follow the power savings recommendations listed in the "Power consumption versus performance" section.

Optimizing storage I/O performance

To achieve excellent system performance, maintaining an appropriate balance between processing power, memory capacity/performance, crossbar interconnectivity and system I/O capabilities is of paramount importance. Each BL920s Gen8 Server Blade provides the necessary I/O capabilities to maintain that balance. These server blades use a flexible and customer-configurable I/O design that can accommodate two Flexible LAN-on-Motherboard (FlexLOM) devices and three mezzanine card slots. These five configurable component slots connect directly to the E7 v2 processors through multi-channel PCIe Gen3 links, and are capable of providing a maximum I/O bandwidth of up to 100 GB/s per blade. FlexLOM technology and mezzanine cards put the I/O interface on daughter cards, allowing you to choose a specific communication technology while keeping the interfaces closely-coupled to the main system architecture.

To illustrate how to configure the Superdome X server for the best throughput and lowest latencies, the Flexible I/O Tester (FIO) was used to simulate a reasonable Oracle I/O workload on a test environment. FIO software and documentation is available at freecode.com/projects/fio. The Superdome X was then tuned to maximize the throughput and I/O latencies of the system for the same workload.

Test environment

The test environment for the Oracle I/O simulation includes an 8-socket (120 core) Superdome X server connected to a 3PAR 7450 array with all SSD storage drives. The Superdome X used eight fiber channel (FC) ports connected to the 3PAR through the FC Interconnect module, resulting in 16 paths to each virtual LUN.

When testing I/O performance, ensure all disks are running a current version of their respective software/firmware. 3PAR arrays should be running 3PAR OS 3.2.1 (or later) with a current patch set. 3PAR OS updates can be downloaded from the Software Evaluations Ports: http://hp.com/software/evalportal.

I/O simulation

The FIO benchmark tool was used to simulate an Oracle I/O workload consisting of the following job roles:

- 480 readers that perform 8K random reads, similar to Oracle Shadow/Server processes
- 32 writers that perform 8K random writes using libaio, similar to Oracle DB Writers
- Eight parallel_query tasks that perform 1MB sequential reads using libaio, similar to full table scans performed by various processes including Oracle Parallel Query processes
- log_writer that performs variable sized sequential writes to two files in parallel using libaio, where the size of the writes ranges from 512 bytes to 300 kilobytes (KB)

The goal of the FIO benchmark is not to stress the I/O storage subsystems but to see how the various recommendations impact the I/O latencies and overall throughput of the simulated workload. The FIO profile used to generate the above workload can be found in the Appendix: FIO profile used in testing.

Test configurations

Two different configurations were used for FIO testing, as shown in Table 8.The first configuration is a Low Performance configuration where throughput results and latencies were expected to be non-optimal. Many of these tunable values in the Low Performance configuration are default values or former RHEL release default values. The second configuration is the High Performance configuration, with tunable values optimized for this specific test and configuration.

Table 8. FIO test configurations.

Tunable	Low-performance configuration	High-performance configuration
/sys/block/*/queue/scheduler	cfq	deadline
/sys/block/*/queue/nr_requests	128	1024
/sys/block/*/queue/max_sectors_kb	512 KB	512 KB
/sys/block/*/queue/rotational	1	0
/sys/block/*/queue/nomerges	0	1
/sys/block/*/device/queue_depth	32	128
/sys/block/*/device/scsi_disk/*/cache_type [1]	write back	write through
sysctl kernel.numa_balancing [1]	1	0
tuned-adm profile	balanced	latency-performance
cpupower frequency-set –g	powersave	N/A
rr_min_io_rq (/etc/multipath.conf)	1	1

Note:
[1] Valid only for RHEL.

Test results

Using the same FIO tests on the same test environment, the High Performance configuration resulted in an 8% improvement in throughput and a 5% improvement in I/O latencies (Table 9).

Table 9. FIO test results.

Measurement	Low-performance configuration	High-performance configuration
Read Throughput	928 MB/s	1002 MB/s
Read IOPS	95816	103446
Read average size	9	9
Read latency	0.61 ms	0.59 ms
Write Throughput	241	273
Write IOPS	22766	27088
Write average size	10	10
Write latency	0.43 ms	0.41 ms

Note that every system is different and every I/O benchmark is unique. Your results may vary. In general, the High Performance configuration will yield the best performing I/O in most cases.

Explanation of tunable settings used for FIO testing

Three type of parameters are involved in the FIO tests; block device, SCSI, and multi-path.

Block Device parameters
Block device parameters can be altered by echoing a new value into the appropriate file. For example:

```
$ echo deadline >/sys/block/sdab/queue/scheduler
```

scheduler

The Completely Fair Queuing (CFQ) scheduler is the default I/O scheduler for RHEL 6 and is suitable for a wide variety of applications. While CFQ provides a good compromise between throughput and latency, we recommend the Deadline I/O scheduler, which is becoming more popular and has proven to perform better for database systems and low latency devices. The Deadline I/O scheduler is the new default scheduler for RHEL 7. The noop scheduler should also perform well for database and low latency devices and can be used if desired. The I/O scheduler can also be configured using the elevator=<scheduler> boot command line option or by using the tuned-adm command and selecting the “latency-performance” or “network latency” profile.

max_sectors_kb

The max_sectors_kb controls the maximum size of a physical I/O that can be sent to the SCSI device. The default value is 512 KB. In some environments where large I/O is performed, increasing the parameter to 1024 or even 2048 can reduce the number of physical I/Os, which can be a benefit if the number of I/Os or interrupts are impacting performance. However, in some cases, decreasing the value may be a benefit as a large request could be broken down into several smaller requests and issued in parallel along multiple paths. For example, consider an application that issues 1 MB requests to a device that has 8 FC paths. Setting max_sectors_kb to 128 can allow 1 I/O to be sent in parallel to each of the eight paths, which increases the bandwidth by engaging all eight paths in parallel.

nr_requests

The nr_requests parameter is the number of read requests and the number of write requests allocated for a device. The default value is 128, which means 128 read requests and 128 write requests can be allocated for a device. If all of the device requests are in use for active or queued I/Os, then a task must wait for a free request in order to initiate the I/O. Increasing the nr_requests value allows for more active I/O request.

rotational

The block device schedulers will perform some additional overhead to sort and merge I/O requests to reduce disk head movement. For SSD drives, the rotational value can be set to 0, although very little improvement if any is noticed.

nomerges

If the nomerges tunable is set to 1, the I/O requests are not merged in the block device request queue. This is good for mostly random requests as well as synchronous or direct I/O request. For standard fileystems, using the default value of 0 is sufficient for nomerges.

SCSI parameters

The SCSI device parameters can be altered by echoing a new value into the appropriate file. For example:

```
$ echo 128 >/sys/block/sdab/device/queue_depth
```

queue_depth

While the nr_requests parameter controls how many requests can be queued or in progress to a device, the SCSI queue_depth parameter controls how many I/Os can be issued (in progress) to a device. The default value is 32. If nr_requests is 128 and the queue_depth value is 32, then 32 requests can be issued to the device and another 96 requests can be inserted into the device queue waiting to be issued. For large LUNs with many underlying disk spindles or for low latency devices like SSD, increasing the queue_depth parameter can allow more parallel activity for the device.

cache_type (Barrier Writes)

The purpose of barrier writes it to ensure the on-disk consistency and ordering of critical data. The barrier writes are implemented using the SYNCHRONIZE_CACHE SCSI command. The SYNCHRONIZE_CACHE command forces the disk to flush its internal disk cache to the physical spindles, so that a power outage does not cause any loss of data. However, with more intelligent disk arrays, with battery-backed (non-volatile) cache, barrier writes are typically not needed.

Whether or not the SYNCHRONIZE_CACHE command is issued depends on how the storage array advertises itself. If the array advertises as "write cache disabled" or "write through cache", the SYNCHRONIZE_CACHE command is not issued. Note that some disk arrays with non-volatile caches will advertise as "write cache disabled" even though it does indeed have a write cache. However, if the array advertises as "write cache enabled", also called “writeback cache", the SYNCHRONIZE_CACHE command would be sent for synchronous writes to the device. The write cache behavior can typically be seen in dmesg output during system boot, for example:

sd 7:0:5:0: [sdo] Write cache: enabled, read cache: enabled, supports DPO and FUA

When non-volatile disks that advertise as “write cache: enabled” are used with filesystems, certain operations such as a sync() or journal log flush can cause unnecessary barrier writes. For filesystems, barrier writes can typically be disabled using the barrier=0 mount option. While barriers can be easily disabled with filesystem mount options, the same is not true for synchronous access to non-filesystem block device files, as is used with Oracle Automatic Storage Management (ASM).

Starting with RHEL 6.5 and RHEL 7, the devices "cache_type" can be modified to alter the behavior of a disk device. For example, to change the cache_type of a device to avoid the barrier writes, the device can be changed from "write back" to "write through" by writing the value "temporary write through" to the cache_type file.

```
$ cat  /sys/devices/pci0000:00/0000:00:05.0/0000:11:00.0/host3/rport-3:0-
5/target3:0:5/3:0:5:0/scsi_disk/3:0:5:0/cache_type
write back
$ echo "temporary write through" >
/sys/devices/pci0000:00/0000:00:05.0/0000:11:00.0/host3/rport-3:0-
5/target3:0:5/3:0:5:0/scsi_disk/3:0:5:0/cache_type
$ echo 1 > /sys/devices/pci0000:00/0000:00:05.0/0000:11:00.0/host3/rport-3:0-
5/target3:0:5/3:0:5:0/rescan
$ cat  /sys/devices/pci0000:00/0000:00:05.0/0000:11:00.0/host3/rport-3:0-
5/target3:0:5/3:0:5:0/scsi_disk/3:0:5:0/cache_type
write through
```

Note

The device must be rescanned for the change to take effect and is only in effect until the next reboot. The SCSI block devices will need to be closed and reopened for the change to take effect, which may mean applications (such as Oracle) will need to be restarted. Also, if you are using the multipath driver, you can reconfigure the devices through the `multipathd` command as the SCSI block devices are opened by the following multipath daemon:

```
$ multipathd reconfigure
```

Note
3PAR storage arrays originally advertised as "Write cache: enabled." With 3PAR OS version 3.2.1 GA/MU1 or later, the 3PAR device should advertise as "Write cache: disabled", preventing the unnecessary barrier writes.

WARNING
Disabling barrier writes on disk devices with volatile cache can result in data loss or data corruption.

Multipath configuration

The multipath configuration is specified in the /etc/multipath.conf file. Each array has a recommended set of multipath settings based on OS version and whether or not Asymmetrical Logical Unit Assignment (ALUA) is supported. For example, following configuration set[1] is for 3PAR arrays connected to systems running RHEL 7 using ALUA:

```
defaults {
        find_multipaths         yes
        user_friendly_names     yes
        polling_interval        10
}
devices {
        device {
        vendor                  "3PARdata"
        product                 "VV"
        features                "0"
        hardware_handler        "1 alua"
        path_selector           "round-robin 0"
        path_grouping_policy    "group_by_prio"
        prio                    "alua"
        failback                "immediate"
        rr_weight               "uniform"
        no_path_retry           18
        rr_min_io_rq            1
        path_checker            "tur"
        detect_prio             "yes"
        }
}
```

To maximize throughput, be sure to spread the I/O workload over multiple Fibre Channel paths across multiple blades in the Superdome X server. Most storage devices work well with a round-robin path selector and an rr_min_io_rq value of 1. However, different storage devices may behave differently. Contact your storage vendor for the best multipath profile for your storage device.

I/O interrupts

For best results, the I/O interrupts should be processed on the same socket or blade where the I/O card is installed. For the HP QMH2672 FC card, the irqbalance daemon provides appropriate interrupt assignments.

Power savings

To reduce I/O latencies, be sure to follow the power savings recommendations listed in the "Power consumption versus performance" section.

NUMA balancing

By default, NUMA balancing is enabled on RHEL 7. The FIO tests performed better with NUMA balancing disabled (refer to the "Automatic NUMA Balancing" section).

[1] Configuration set source: "HP 3PAR Red Hat Enterprise Linux and Oracle Linux Implementation Guide" found at: hp.com/km-ext/kmcsdirect/emr_na-c04448818-4.pdf

Optimizing Oracle performance

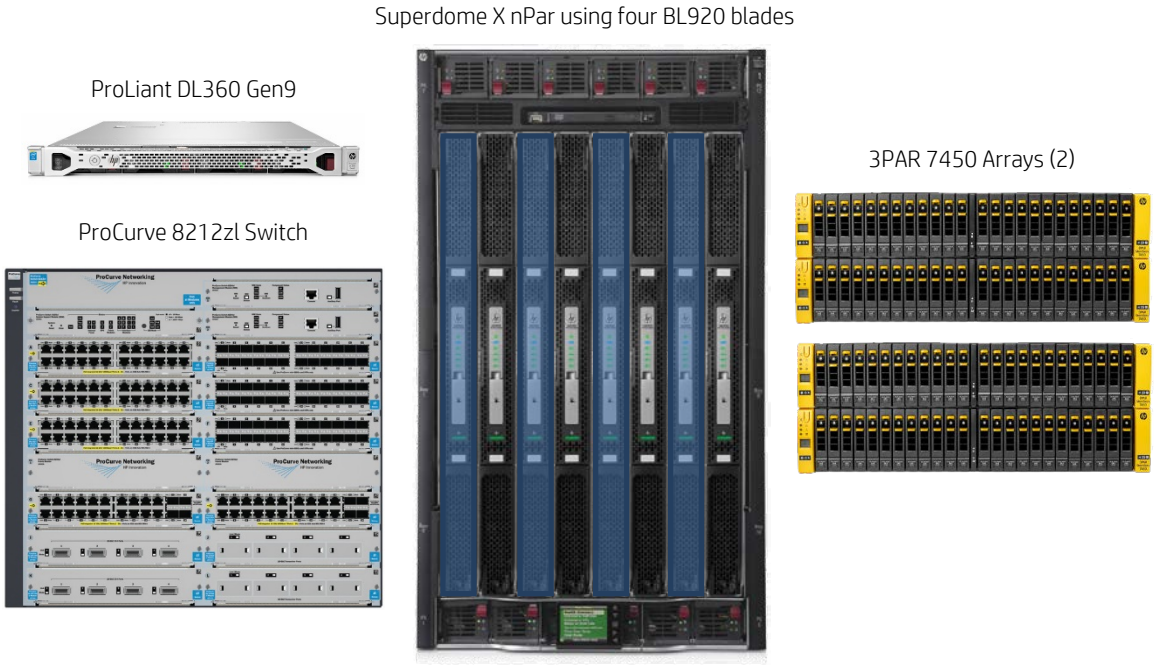
Oracle publishes updated versions of the Database Performance Tuning Guide and Database Administrator's Reference Guide with each new release of their database software. Both guides discuss methods of configuring Oracle instances for optimal performance. This paper does not reproduce what has been covered elsewhere, but instead demonstrates how enabling the previously documented best practices actually affects a typical online transaction processing (OLTP) workload running on large-scale NUMA systems like Superdome X. This section illustrates how a significant increase in Oracle Transactions per Minute (TPM) might be achieved when each of these performance recommendations are implemented.

As with any tuning recommendations, the standard disclaimers apply—every OLTP application workload is different, so the actual amount of improvement seen in your production environment may vary from the results documented here. The goal here is to highlight many of the performance best practices and illustrate the difference in Oracle TPM seen in HP's test environment when different combinations of recommendations are enabled.

Test environment

The components for the Oracle test environment using HammerDB are shown in Figure 3. A four-blade nPar on the Superdome X was used as the Oracle database server. Each blade was fully populated with two 15-core CPUs and 1.5TB of RAM for a total of 120 CPU cores and 6TB of RAM for the nPar. Hyperthreading was enabled on the CPUs during all tests. The database files were stored in ASM disk groups, which were housed on two 3PAR 7450 All-Flash arrays. A ProLiant DL360 Gen9 server was used as the driver system running HammerDB. The DL360 and Superdome X were connected via 10GbE via a ProCurve 8212zl switch.

Figure 3. Oracle test environment.



The HammerDB configuration remained consistent during each test run to more accurately gauge the difference in Oracle TPM resulting from the individual tuning changes. The database schema was built using the TPCC profile with 2000 warehouses and a partitioned Order Line table. Each test involved 200 virtual HammerDB users with zero key and think time. Each test ran for five minutes and Oracle automatic workload repository (AWR) reports were collected after each run to determine the resulting TPM numbers.

Testing was conducted using the following scenarios:

- NUMA optimization
- HugePages
- Multiple Oracle listener processes

Scenario 1: NUMA optimizations

Most modern database applications, including Oracle, are designed to take advantage of the NUMA features of modern scale-up architecture systems like Superdome X. However, in order for applications to take full advantage of the NUMA benefits, the system must be configured properly at all layers of the stack. This means the hardware must be configured properly, the operating system needs to support NUMA features, and in some cases, the application needs to be told specifically to enable NUMA optimizations. Oracle is no exception.

Operating system NUMA configuration

For this test environment, we installed RHEL 7 using the default settings. We then installed Oracle 12c using the instructions provided in the Oracle installation guide. Once Oracle was up and running we did a cursory check to see whether NUMA was enabled at the OS layer by using the numactl(8) command:

```
$ numactl –hardware
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 ... 231 232 233 234 235 236 237 238 239
node 0 size: 6291339 MB
node 0 free: 3078837 MB
node distances:
node   0
  0:  10
```

The Superdome X nPar running Oracle contains four blades with each blade containing two CPU sockets, which means the system *should* be reporting eight distinct NUMA nodes. However, during system boot, only a single NUMA node was created and all CPU cores and hyperthreads were added to that NUMA node. This was confirmed with the following dmesg output from the system boot:

```
kernel: NUMA turned off
kernel: Faking a node at [mem 0x0000000000000000-0x0000061fffffffff]
kernel: Initmem setup node 0 [mem 0x00000000-0x61fffffffff]
```

This indicates NUMA was disabled during system boot. A quick check of the /proc/cmdline file reveals the following parameter settings that were passed to the kernel during boot:

```
$ cat /proc/cmdline
BOOT_IMAGE=/vmlinuz-3.10.0-210.el7.x86_64 root=/dev/mapper/rhel-root ro rd.lvm.lv=rhel/root
crashkernel=512M rd.lvm.lv=rhel/swap vconsole.font=latarcyrheb-sun16 vconsole.keymap=us
nosplash earlycon=uart8250,io,0x3f8,115200n8 intel_idle.max_cstate=1 numa=off
console=ttyS0,115200
```

So where did this “numa=off” parameter come from?

As it turns out, Oracle provides RPMs to simplify the installation of Oracle 11g and 12c database software. The RPMs are named “oracle-rdbms-server-11gR2-preinstall” and “oracle-rdbms-server-12cR1-preinstall” respectively. These RPMs perform several tasks, including; creating the “oracle” user and associated groups, making necessary adjustments to various kernel parameters for subsystems such as shared memory, and sets shell resource limits. They may also add “numa=off” to the kernel boot string, which disables all NUMA features in the kernel. This will result in sub-optimal memory allocations and task scheduling decisions by the Linux kernel, resulting in poor performance, especially on scale-up platforms such as Superdome X.

Recommendation

Remove the “numa=off” parameter from the kernel boot syntax and reboot the server.

Oracle NUMA optimizations

Once the nPar was rebooted with the “numa=off” parameter removed from the boot string, the Oracle instance was started. By monitoring the alert.log file for the instance, we can determine whether Oracle is running in NUMA mode or not:

```
NUMA system with 8 nodes detected
Oracle NUMA support not enabled
The parameter _enable_NUMA_support should be set to TRUE to enable Oracle NUMA support
```

The highlighted alert.log text confirms Oracle now detects it is running on a NUMA-capable platform but Oracle itself is not configured to run in NUMA mode. The Oracle instance was stopped and the **"_enable_NUMA_support=TRUE"** parameter was added to the database instance configuration file (PFILE or SPFILE). Upon re-starting the instance, the alert.log now shows:

```
NUMA system with 8 nodes detected
Oracle NUMA support enabled
NUMA node details:
  OS NUMA node 3 (30 cpus) mapped to Oracle NUMA node 0
  OS NUMA node 4 (30 cpus) mapped to Oracle NUMA node 1
  OS NUMA node 5 (30 cpus) mapped to Oracle NUMA node 2
  OS NUMA node 6 (30 cpus) mapped to Oracle NUMA node 3
  OS NUMA node 7 (30 cpus) mapped to Oracle NUMA node 4
  OS NUMA node 0 (30 cpus) mapped to Oracle NUMA node 5
  OS NUMA node 1 (30 cpus) mapped to Oracle NUMA node 6
  OS NUMA node 2 (30 cpus) mapped to Oracle NUMA node 7
```

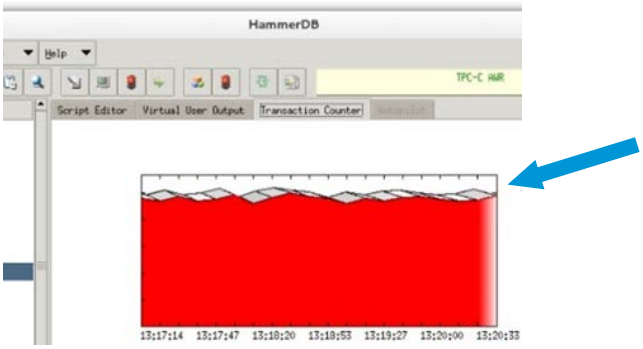
Recommendation

Configure the Oracle instance with the "_enable_NUMA_support=TRUE" parameter.

Automatic NUMA balancing

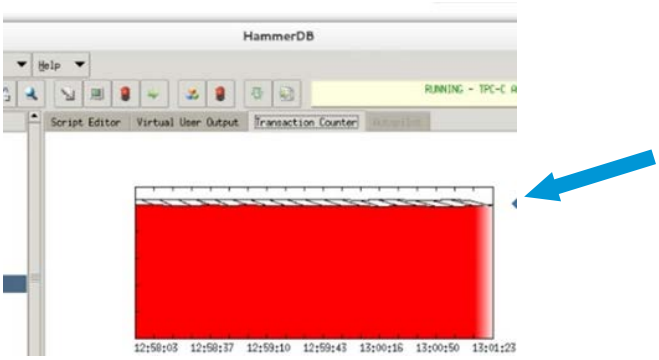
With the OS and Oracle instances both configured for NUMA support, an initial HammerDB test was started. This initial run showed several anomalies. First, the overall TPM numbers were lower than expected. Second, the Oracle transactions fluctuated constantly during the test period, resulting in a transactional graph similar to the one shown in Figure 4.

Figure 4. Oracle transactions in HammerDB run with Automatic NUMA balancing enabled.



By comparison, a normal HammerDB run should result in a flat and consistent transactional counter graph (Figure 5).

Figure 5. Oracle transactions in HammerDB run with expected transactional counter graph.



Note

The TPM counter and scale numbers normally present during a HammerDB run are not shown in these figures.

In addition to the low TPM values and erratic nature of the HammerDB transaction counter, the "Top 10 Foreground Events by Total Wait Time" data in the resulting Oracle AWR report contained unexpected values in the "% DB time" column (Table 10):

Table 10. Top 10 Foreground Events by Total Wait Time reported by Oracle AWR.

Event	Waits	Total wait time (sec)	Wait average (ms)	% DB time	Wait class
DB CPU		29.3K		57.6	
library cache: mutex X	862,670	1458.5	1.69	2.9	Concurrency
latch: enqueue hash chains	261,074	1438	5.51	2.8	Other
log file sync	25,302	203.7	8.05	.4	Commit
db file scattered read	99,039	186	1.88	.4	User I/O
latch: In memory undo latch	387,814	163.2	0.42	.3	Concurrency
enq: TX - row lock contention	56,271	131.2	2.33	.3	Application
db file sequential read	77,502	114.8	1.48	.2	User I/O
latch: undo global data	12,188	57.6	4.72	.1	Other
buffer busy waits	103,323	39.9	0.39	.1	Concurrency

Ordinarily, the numbers in the "% DB time" column typically add up to a value approximating 100%. In this test, the DB time percentage numbers added up to roughly 65%.

The anomalies were investigated further by using the Linux KI toolset developed by HP that provides kernel tracking and reporting functions. LinuxKI can either use the FTRACE tracing facility built into the Linux kernel or an HP written opensource DLKM called LiKI which collects a more feature rich set of kernel trace events. A user space report tool called kiinfo can report on the trace data collected from either FTRACE or LiKI. The LinuxKI toolset is delivered via RPM or dpkg. The LinuxKI toolset use is restricted to Hewlett-Packard servers. Please contact your Hewlett-Packard representative for more information.

A new HammerDB run was started and during the run a 20-second Linux KI collection was captured for analysis. The Linux KI output helps to answer the following questions pertinent for any performance engagement:

- When the application is running, what is it doing?
- When the application is sleeping, what is it waiting on?

In this case, we are interested in knowing what the Oracle server (or shadow) processes are doing when they are executing on a CPU, and what they are waiting on when they sleep. One of the Oracle shadow processes was selected for analysis and the first thing to note is how much time the process actually spent running, sleeping, and waiting on the run queue. This information is presented in the "SCHEDULER ACTIVITY REPORT" section of the Linux KI output, which is generated for each process. In the following example, the Oracle process was found to be spending nearly 93% of its time sleeping (18.56 seconds of a 20 second data collection period).

```
PID 150089  oracleNUKYD1
  PPID 1  /usr/lib/systemd/system

    ********* SCHEDULER ACTIVITY REPORT ********
    RunTime    :  1.240211  SysTime   :  0.300939   UserTime   :  0.939272
    SleepTime  : 18.566175  Sleep Cnt :     11321   Wakeup Cnt :       629
    RunQTime   :  0.182789  Switch Cnt:     11714   PreemptCnt :       393
    Last CPU   :        43  CPU Migrs :      4330   NODE Migrs :       858
    Policy     : SCHED_NORMAL     vss :  27916259         rss :      7205
    busy   :      6.20%
      sys  :      1.51%
```

```
      user :      4.70%
     runQ  :      0.91%
     sleep :     92.88%
```

Knowing that the process spends most of the time sleeping, the next question is; "What is it waiting on?" In the "Kernel Functions calling sleep()" section of the Linux KI output (shown below), we find that the process was spending the highest percentage of time (Slp%) sleeping in the **sleep_on_page()** routine.

```
Kernel Functions calling sleep() - Top 20 Functions
   Count     Pct    SlpTime    Slp% TotalTime%   Msec/Slp   MaxMsecs  Func
     806   7.12%     5.9461  32.03%     29.75%      7.377     80.777  sleep_on_page
    1669  14.74%     5.5614  29.95%     27.82%      3.332     75.438  do_blockdev_direct_IO
    6728  59.43%     3.9617  21.34%     19.82%      0.589    159.637  sk_wait_data
    1958  17.30%     2.7493  14.81%     13.75%      1.404    139.797  poll_schedule_timeout
     126   1.11%     0.2368   1.28%      1.18%      1.879     73.157  read_events
      32   0.28%     0.1013   0.55%      0.51%      3.167     10.763  __mutex_lock_slowpath
       1   0.01%     0.0095   0.05%      0.05%      9.541      9.541  sleep_on_page_killable
```

Once we know the function where the Oracle process is spending the majority of its time sleeping, we can look at the process stack trace information to see how the process ended up eventually calling sleep_on_page(). Linux KI collects the stack trace records for every running process on the system. These records are collected every time a process sleeps, and they show the successive series of kernel routines the process called just prior to sleeping. From this information we hope to better understand what the process was trying to do when it was forced to sleep and wait in the sleep_on_page() routine.

Further review of the stack trace that eventually calls sleep_on_page() reveals the process is handling a page fault. We know this because **page_fault()** is the first system call at the tail end of the highlighted stack trace. By analyzing the various system calls in the stack trace and reviewing the source code for these system calls, we deduce the process is going to sleep waiting on a memory page migration from one NUMA node to another.

```
Process Sleep stack traces (sort by % of total wait time) - Top 20 stack traces
    count    wpct      avg   Stack trace
               %     msecs
================================================================
     1668  29.90     3.328   do_blockdev_direct_IO  __blockdev_direct_IO  blkdev_direct_IO
generic_file_aio_read  do_sync_read  vfs_read  sys_pread64  tracesys  |  __pread_nocancel
ksfd_skgfqio  ksfd_io  ksfdread  kcfrbd1  kcbzib  kcbgtcr
     6621  21.33     0.598   sk_wait_data  tcp_recvmsg  inet_recvmsg  sock_aio_read.part.7
sock_aio_read  do_sync_read  vfs_read  sys_read  tracesys  |  __read_nocancel  nttfprd
nsbasic_brc  nsbrecv  nioqrc  opikndf2
     1907  14.16     1.379   poll_schedule_timeout  do_sys_poll  sys_poll  tracesys  |
__poll_nocancel  sskgxp_selectex  skgxpiwait  skgxpwaiti  skgxpwait  ksxpwait  ksliwat
kslwaitctx  ksxprcv_int  ksxprcvimdwctx  kclwcrs
      100   4.49     8.332   sleep_on_page  __wait_on_bit  wait_on_page_bit
__migration_entry_wait.isra.37  migration_entry_wait  handle_mm_fault  __do_page_fault
do_page_fault  page_fault  |  lxeg2u  ldxdts  evadis  evaopn2  qerixGetKey  qerixStart
       84   3.53     7.805   sleep_on_page  __wait_on_bit  wait_on_page_bit
__migration_entry_wait.isra.37  migration_entry_wait  handle_mm_fault  __do_page_fault
do_page_fault  page_fault  |  lxeg2u  ldxdts  evadis  evaopn2  qerixGetKey  qerixStart
```

Now that we have identified the Oracle process stack trace of interest, we can look at the "non-idle GLOBAL HARDCLOCK STACK TRACES" report to see how often all processes on the system are executing a similar stack trace. Linux KI gathers what are called "HARDCLOCK" records from every CPU core and hyperthread every 10 milliseconds. HARDCLOCK trace records include the state of the processor as well as a kernel stack trace if the processor was executing kernel code. These HARDCLOCK records help answer the standard performance question for the system—"If it's running, what is it doing?"

A quick look at the HARDCLOCK report (shown below) indicates the *entire system* is spending some amount of time migrating memory pages on behalf of running processes:

```
non-idle GLOBAL HARDCLOCK STACK TRACES (sort by count):

   Count     Pct  Stack trace
============================================================
     416   0.82%  get_gendisk  blkdev_get  raw_open  chrdev_open  do_dentry_open
finish_open  do_last  path_openat  do_filp_open  do_sys_open  sys_open  tracesys
     382   0.75%  __blk_run_queue  __elv_add_request  blk_insert_cloned_request
dm_dispatch_request  dm_request_fn  __blk_run_queue  queue_unplugged  blk_flush_plug_list
blk_finish_plug  do_blockdev_direct_IO  __blockd
ev_direct_IO  blkdev_direct_IO  generic_file_aio_read  do_sync_read  vfs_read  sys_pread64
     379   0.74%  remove_migration_pte  rmap_walk  migrate_pages  migrate_misplaced_page
do_numa_page  handle_mm_fault  __do_page_fault  do_page_fault  page_fault
     362   0.71%  __page_check_address  try_to_unmap_one  try_to_unmap_file  try_to_unmap
migrate_pages  migrate_misplaced_page  do_numa_page  handle_mm_fault  __do_page_fault
do_page_fault  page_fault
     344   0.67%  __mutex_lock_slowpath  mutex_lock  try_to_unmap_file  try_to_unmap
migrate_pages  migrate_misplaced_page  do_numa_page  handle_mm_fault  __do_page_fault
do_page_fault  page_fault
```

This raises the question: "Why are all of these memory migrations occurring?" As described earlier, Red Hat Enterprise Linux version 7 introduced a new feature called "Automatic NUMA Balancing." This feature was added in an attempt to make the system more "NUMA-aware," and instructs the kernel to migrate a running task's memory pages to the same NUMA node where the task is executing. The purpose is to keep associated memory pages in close proximity to the CPU cores running the process, resulting in a reduction in memory access latencies and increased performance.

For many applications this new behavior does in fact improve performance. However, due to the size of the Oracle system global area (SGA) used in these tests and the frequency of task migrations between NUMA nodes, this migration of memory results in a significant decrease in TPM.

Recommendation
Disable the Automatic NUMA Balancing feature by setting the "kernel.numa_balancing" parameter to 0 via the command "sysctl kernel.numa_balancing=0".

With all of the NUMA features correctly configured at the OS and application layer, the resulting TPM numbers ***increased by an average of 27%*** in our test environment.

This is a good reminder that not all "performance related" features will benefit every application. This scenario also serves as an excellent demonstration of the power of the Linux KI toolset and why it is an invaluable resource in identifying the underlying cause of performance problems.

Scenario 2: HugePages

One of the core functions of the Linux kernel (and most modern operating systems) is managing the memory resources of the system. On systems that support virtual memory addressing, the kernel must maintain a mapping between the virtual memory address space and the physical memory where data is stored. These mappings are stored in a page table, with each individual mapping referred to as a page table entry. By default the Linux kernel allocates memory using a page size of 4KB. On large memory systems, such as Superdome X, the number of page table entries required to map these relatively small 4KB memory pages can be significant.

In order to reduce the overhead of virtual-to-physical memory translations, the memory management unit of each CPU maintains a cache of recently accessed translations called the Translation Lookaside Buffer (TLB). The TLB is a fixed size cache. It is reasonable to conclude that as the amount of physical memory in the server grows and the size of memory pages remains relatively small, the number of page table entries increases to the point where the chances of a process finding the virtual memory translation it needs in the TLB cache is remote.

The HugePages feature was created to address this problem. It allows the kernel to support much larger sized memory pages, varying from two megabytes and above. Using larger page sizes can improve performance by reducing the amount of system resources required to manage and access page table entries. The use of HugePages means the overall size of the page table is reduced since each page table entry now points to a 2MB (or larger) memory page. With fewer page table

entries to manage, the chances of successfully finding the virtual-to-physical translation your process needs in the TLB cache increases, resulting in improved performance.

Another benefit to using HugePages with Oracle databases is the memory associated with these large pages are locked in memory and never swapped out, thus forcing the Oracle SGA to remain memory resident.

HugePages configuration

Configuring an Oracle instance to use HugePages requires the following procedure:

- Configure the "memlock" setting in the /etc/security/limits.conf file
- Determine the appropriate number of HugePages to allocate based on SGA size
- Configure the desired HugePage pool in the /etc/sysctl.conf file
- Confirm the HugePage pool is created and sized correctly
- Ensure the Oracle parameter "USE_LARGE_PAGES" is set to either "TRUE" or "ONLY"
- Disable Automatic Memory Management for the Oracle instance
- Disable Transparent HugePages

All of the above steps are described in Appendix G of the current Oracle Database Administrator's Reference Guide.

Confirming Oracle's use of HugePages

To confirm whether the Oracle instance is correctly using memory from the HugePage pool to house the SGA, monitor the Oracle alert.log file during instance startup. If the SGA is using the HugePage pool then the alert.log should contain a section similar the one shown below.

```
************************************************************************
Dump of system resources acquired for SHARED GLOBAL AREA (SGA)

Per process system memlock (soft) limit = UNLIMITED
Expected per process system memlock (soft) limit to lock
 SHARED GLOBAL AREA (SGA) into memory: 2104G
Available system pagesizes:
  4K, 2048K
Supported system pagesize(s):
  PAGESIZE  AVAILABLE_PAGES  EXPECTED_PAGES  ALLOCATED_PAGES  ERROR(s)

     2048K          1536000         1077250         1077250        NONE
Reason for not supporting certain system pagesizes:
  4K - Large pagesizes only
************************************************************************
```

The above highlighted text shows the SGA is being allocated from a memory pool whose page size is 2048K (2MB), thus confirming the SGA is being allocated from the HugePage pool. The number of pages in the HugePage pool in this example is 1,536,000, indicating the HugePage pool contains 3TB of memory. The SGA for this instance is consuming only 1,077,250 of these pages. This could mean the HugePage pool is oversized for this system, or that other applications and Oracle instances have not yet been started, and those applications will also make use of the HugePage pool.

Recommendation

Configure the Oracle instance to use HugePages of 2MB.

With the Oracle instance configured to use HugePages, the resulting TPM numbers increased *by an average of 13%* in our test environment.

In addition to the increase in Oracle TPM, the CPUs spent an average of 7% less time executing "system" code when HugePages were enabled. This reduction in system time can be attributed to the reduction in the size and management of the page table and the resulting increased TLB hit rate.

To quantify the effect of enabling HugePages on the size of the system page table, the "PageTables" parameter reported via /proc/meminfo was monitored during HammerDB runs with HugePages enabled and disabled.

HugePages Disabled:

```
 # grep PageTables /proc/meminfo
PageTables:      5556296 kB
```

HugePages Enabled:

```
# grep PageTables /proc/meminfo
PageTables:       175224 kB
```

The page table size decreased from 5.5GB to 175MB by enabling HugePages for this single Oracle instance. Keep in mind this test involves only 200 users, and page table entries are a per-user construct, so an Oracle environment involving more than 200 active connections to a database instance will likely see an even larger difference in page table size, and potentially a larger increase in TPM.

Scenario 3: Multiple Oracle Listener Processes

Now that the Oracle instance is using HugePages to store the SGA and the NUMA configuration has been properly configured (Automatic NUMA Balancing is disabled and Oracle NUMA optimizations are enabled), a new HammerDB run was started and a fresh Linux KI collection was captured and analyzed.

The "Global CPU Usage by LDOM" section of the Linux KI report provides a high-level view of the total system CPU utilization, broken down by Logical Domain (LDOM, and this being a NUMA platform, also referred to as NUMA node) and the category of work the CPUs in that LDOM were performing. The overview (below) shows a major imbalance in the CPU utilization across the LDOMs/NUMA nodes:

```
node  ncpu     Total Busy          sys          usr         idle
   0 [ 30] :       99.84%        6.82%       93.02%        0.16%
   1 [ 30] :       54.96%        4.21%       50.75%       45.04%
   2 [ 30] :       52.06%        4.05%       48.01%       47.94%
   3 [ 30] :       46.16%        3.61%       42.55%       53.84%
   4 [ 30] :       51.22%        4.26%       46.96%       48.78%
   5 [ 30] :       50.43%        3.94%       46.49%       49.57%
   6 [ 30] :       56.25%        4.40%       51.86%       43.75%
   7 [ 30] :       51.85%        3.97%       47.88%       48.15%
Total              57.86%        4.41%       53.45%       42.14%
```

The CPU cores and hyperthreads comprising NUMA node 0 are nearly 100% busy while other NUMA nodes are roughly 50% busy. What is causing this imbalance of resources? Table 11 shows the detailed HARDCLOCK data of what the CPUs were doing at each clock tick. This data reveals that NUMA node 0 is servicing a much higher portion of the interrupts (bolded text in table 11) than other nodes:

Table 11. Detailed CPU usage by NUMA node.

NUMA node	User%	Sys%	Intr%	Idle%
0	84.54%	8.47%	**6.98%**	0.01%
1	82.76%	11.04%	0.47%	5.73%
2	81.97%	11.09%	0.56%	6.38%
3	82.19%	11.19%	0.56%	6.07%
4	82.31%	11.55%	0.66%	5.48%
5	82.24%	10.70%	0.55%	6.51%
6	83.29%	10.35%	0.60%	5.76%
7	82.45%	11.43%	0.65%	5.48%

This imbalance of resources is due to the single Oracle listener process being bound to the IP address associated with the network interface in NUMA node 0, and the interrupts for this network card are mapped to the CPU cores and hyperthreads in NUMA node 0. The Linux scheduler attempts to move tasks to the NUMA node where the interrupts associated with the Oracle request are being serviced. Since this is the only network interface receiving Oracle requests, NUMA node 0 ends up not only processing more Oracle requests, it also processes the bulk of the system interrupts.

The method described in the following sections spreads the workload among the NUMA nodes evenly by:

- Starting multiple Oracle listener processes (one per NUMA node)
- Associating each listener with a different physical network interface
- Distributing the Oracle requests evenly among the multiple listener processes

Network interface configuration

The first step is to configure each network interface in a different subnet. This ensures that the same network interface that receives the requests will be used to send the outbound reply. In this example, each of the Superdome X LOM ports were configured with a different subnet.

```
[root@sdx ~]# ip addr | grep -E 'eno|inet' | grep -vE '127|inet6'
2: eno24: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    inet 11.1.1.1/24 brd 11.1.1.255 scope global eno24
3: eno25: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    inet 12.1.1.1/24 brd 12.1.1.255 scope global eno25
4: eno200: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    inet 13.1.1.1/24 brd 13.1.1.255 scope global eno200
5: eno201: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    inet 14.1.1.1/24 brd 14.1.1.255 scope global eno201
6: eno376: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    inet 15.1.1.1/24 brd 15.1.1.255 scope global eno376
7: eno377: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    inet 16.1.1.1/24 brd 16.1.1.255 scope global eno377
8: eno552: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    inet 17.1.1.1/24 brd 17.1.1.255 scope global eno552
9: eno553: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    inet 18.1.1.1/24 brd 18.1.1.255 scope global eno553
```

Listener configuration

The next step is to configure the Oracle listeners. There are several ways to configure listener processes but in this example the following entries were added to the “listener.ora” file. Each listener is assigned a unique name, configured to one of the eight unique IP addresses, and bound to a unique TCP/IP port number.

```
ORCL_1608 =
  (DESCRIPTION_LIST =
    (DESCRIPTION =
      (ADDRESS = (PROTOCOL = TCP)(HOST = 18.1.1.1)(PORT = 1608))
    )
  )
ORCL_1607 =
  (DESCRIPTION_LIST =
    (DESCRIPTION =
      (ADDRESS = (PROTOCOL = TCP)(HOST = 17.1.1.1)(PORT = 1607))
    )
  )

ORCL_1606 =
  (DESCRIPTION_LIST =
    (DESCRIPTION =
      (ADDRESS = (PROTOCOL = TCP)(HOST = 16.1.1.1)(PORT = 1606))
    )
  )
ORCL_1605 =
  (DESCRIPTION_LIST =
    (DESCRIPTION =
      (ADDRESS = (PROTOCOL = TCP)(HOST = 15.1.1.1)(PORT = 1605))
```

```
    )
  )
ORCL_1604 =
  (DESCRIPTION_LIST =
    (DESCRIPTION =
      (ADDRESS = (PROTOCOL = TCP)(HOST = 14.1.1.1)(PORT = 1604))
    )
  )
ORCL_1603 =
  (DESCRIPTION_LIST =
    (DESCRIPTION =
      (ADDRESS = (PROTOCOL = TCP)(HOST = 13.1.1.1)(PORT = 1603))
    )
  )
ORCL_1602 =
  (DESCRIPTION_LIST =
    (DESCRIPTION =
      (ADDRESS = (PROTOCOL = TCP)(HOST = 12.1.1.1)(PORT = 1602))
    )
  )
ORCL_1601 =
  (DESCRIPTION_LIST =
    (DESCRIPTION =
      (ADDRESS = (PROTOCOL = TCP)(HOST = 11.1.1.1)(PORT = 1601))
    )
  )
```

Next, we need to start each of these listener processes. Since we want each of these listener processes bound to a specific NUMA node, the numactl(8) command is used to forcibly associate each individual listener process to the NUMA node where their respective network interface resides. If the numactl(8) command is not available on your system it can be installed via YUM, YAST, rpm, etc.

```
/bin/numactl --cpunodebind=0 /oracle/product/12.1.0/grid/bin/lsnrctl start ORCL_1601
/bin/numactl --cpunodebind=1 /oracle/product/12.1.0/grid/bin/lsnrctl start ORCL_1602
/bin/numactl --cpunodebind=2 /oracle/product/12.1.0/grid/bin/lsnrctl start ORCL_1603
/bin/numactl --cpunodebind=3 /oracle/product/12.1.0/grid/bin/lsnrctl start ORCL_1604
/bin/numactl --cpunodebind=4 /oracle/product/12.1.0/grid/bin/lsnrctl start ORCL_1605
/bin/numactl --cpunodebind=5 /oracle/product/12.1.0/grid/bin/lsnrctl start ORCL_1606
/bin/numactl --cpunodebind=6 /oracle/product/12.1.0/grid/bin/lsnrctl start ORCL_1607
/bin/numactl --cpunodebind=7 /oracle/product/12.1.0/grid/bin/lsnrctl start ORCL_1608
```

Oracle parameters

With the Oracle listener processes running, each bound to their respective network interface and NUMA node, the Oracle instance needs to be modified to use the new listener processes. This can be done via the init.ora parameters, whether that involves using a PFILE or an SPFILE.

```
local_listener='ORCL_1601'
local_listener='ORCL_1602'
local_listener='ORCL_1603'
local_listener='ORCL_1604'
local_listener='ORCL_1605'
local_listener='ORCL_1606'
local_listener='ORCL_1607'
local_listener='ORCL_1608'
```

HammerDB driver system configuration

With the Oracle instance now configured to use multiple listeners, the remote driver system running HammerDB needs to be configured to send requests to these eight separate listener processes. This can be accomplished by configuring a multi-address entry in the "tnsnames.ora" file. The "LOAD_BALANCE" directive tells Oracle to distribute requests among the configured addresses and port numbers.

```
ORCLML =
  (DESCRIPTION =
    (ADDRESS_LIST =
      (LOAD_BALANCE = on)
      (FAILOVER = off)
      (ADDRESS = (PROTOCOL = TCP)(HOST = 11.1.1.1)(PORT = 1601))
      (ADDRESS = (PROTOCOL = TCP)(HOST = 12.1.1.1)(PORT = 1602))
      (ADDRESS = (PROTOCOL = TCP)(HOST = 13.1.1.1)(PORT = 1603))
      (ADDRESS = (PROTOCOL = TCP)(HOST = 14.1.1.1)(PORT = 1604))
      (ADDRESS = (PROTOCOL = TCP)(HOST = 15.1.1.1)(PORT = 1605))
      (ADDRESS = (PROTOCOL = TCP)(HOST = 16.1.1.1)(PORT = 1606))
      (ADDRESS = (PROTOCOL = TCP)(HOST = 17.1.1.1)(PORT = 1607))
      (ADDRESS = (PROTOCOL = TCP)(HOST = 18.1.1.1)(PORT = 1608))
    )
    (CONNECT_DATA =
      (SERVER = DEDICATED)
      (SERVICE_NAME = orclml)
    )
  )
```

With the above configuration in place, a quick `tnsping` command to the multi-listener service will verify connectivity to the various listener processes as shown below:

```
# tnsping orclml
Used TNSNAMES adapter to resolve the alias
Attempting to contact (DESCRIPTION = (ADDRESS_LIST = (LOAD_BALANCE = on) (FAILOVER = off)
(ADDRESS = (PROTOCOL = TCP)(HOST = 11.1.1.1)(PORT = 1601)) (ADDRESS = (PROTOCOL = TCP)(HOST
= 12.1.1.1)(PORT = 1602)) (ADDRESS = (PROTOCOL = TCP)(HOST = 13.1.1.1)(PORT = 1603))
(ADDRESS = (PROTOCOL = TCP)(HOST = 14.1.1.1)(PORT = 1604)) (ADDRESS = (PROTOCOL = TCP)(HOST
= 15.1.1.1)(PORT = 1605)) (ADDRESS = (PROTOCOL = TCP)(HOST = 16.1.1.1)(PORT = 1606))
(ADDRESS = (PROTOCOL = TCP)(HOST = 17.1.1.1)(PORT = 1607)) (ADDRESS = (PROTOCOL = TCP)(HOST
= 18.1.1.1)(PORT = 1608))) (CONNECT_DATA = (SERVER = DEDICATED) (SERVICE_NAME = orcl)))
OK (60 msec)
```

Now that the driver system is properly configured, a new HammerDB run was started. In the new test "ORCLML" was specified as the target instance. With the Oracle instance configured to use multiple listener processes, the resulting TPM numbers increased *by an average of 25%* in our test environment.

In addition to the higher TPM results, reviewing the Linux KI data from the HammerDB run collected using multiple Oracle listeners shows the overall system CPU utilization is significantly higher and the interrupt processing is now much better distributed across the available NUMA nodes.

node	ncpu	Total Busy	sys	usr	idle
0	[30] :	74.99%	5.59%	69.40%	25.01%
1	[30] :	69.08%	5.45%	63.63%	30.92%
2	[30] :	81.93%	5.72%	76.21%	18.07%
3	[30] :	84.04%	6.05%	77.99%	15.96%
4	[30] :	66.87%	5.91%	60.97%	33.13%
5	[30] :	72.50%	5.98%	66.53%	27.50%
6	[30] :	80.94%	6.82%	74.12%	19.06%
7	[30] :	84.36%	6.33%	78.04%	15.64%
Total		**76.84%**	**5.98%**	**70.86%**	**23.16%**

Recommendation

The use of separate listener processes bound to each NUMA node is an effective means of distributing the Oracle workload among the available Superdome X resources.

Summary of Oracle recommendations

Table 12 below summarizes the various Oracle recommendations covered in this paper and their resulting effect on the Oracle TPM results achieved in our test environment.

Table 12. Oracle recommendation summary.

Configuration	NUMA optimizations	HugePages	Multiple listeners	TPM vs previous	TPM vs default
Default	OFF	OFF	OFF		
NUMA	ON	OFF	OFF	27%	27%
HugePages	ON	ON	OFF	13%	44%
Multi-Listener	ON	ON	ON	25%	80%

The “TPM vs. Previous” column shows the incremental increase in TPM percentage as new features are cumulatively added on top of previous features. The “TPM vs. Default” column shows the total increase in TPM percentage when multiple features are enabled compared to the “Default” configuration where none of the recommendations were enabled.

For example, the “HugePages” row shows a gain of 13% additional TPM was seen when “HugePages” and “NUMA” recommendations were enabled compared to a system that only had the “NUMA” configuration changes enabled. This combination of “HugePages” and “NUMA” recommendations showed a 44% increase in TPM above the “Default” configuration where both HugePages and NUMA were disabled.

In our tests, by enabling all of the previously described recommendations we observed an 80% increase in Oracle TPM over the default out-of-the-box configuration.

Summary of recommendations

Table 13 contains a summary of suggested tunable settings for achieving best overall performance with the Superdome X in a Linux environment.

Note
The suggested tuning parameters can be implemented on an as-needed basis. Not all of settings may apply to every application. We recommend testing each suggested optimization for its suitability to your environment.

Table 13. Summary of suggested tunable settings for best overall performance.

Tunable	Recommended setting
Frame MTU size	9000
ixgbe UDP HW RSS mode	SDFN
Bonded NICs	4-port bond, mode 4
Bond xmit_hash_policy	1
ixgbe rx_usecs	0
net.core.rmem_max	16777216
net.core.wmem_max	16777216
net.ipv4.tcp_rmem	4096 87380 16777216
net.ipv4.tcp_wmem	4096 16384 16777216

Table 13. Summary of suggested tunable settings for best overall performance.

Tunable	Recommended setting
tuned-adm profile	latency-performance
cpupower frequency-set –g	performance
Allowed C-states	0, 1
/sys/block/*/queue/scheduler	deadline
/sys/block/*/queue/nr_requests	1024
/sys/block/*/queue/max_sectors_kb	512 KB
/sys/block/*/queue/rotational	0
/sys/block/*/queue/nomerges	1
/sys/block/*/device/queue_depth	128
/sys/block/*/device/scsi_disk/*/cache_type [1]	write through
sysctl kernel.numa_balancing [1]	0
rr_min_io_rq (/etc/multipath.conf)	1

Note:
[1] Valid only for RHEL.

Conclusion

By optimizing the OS and application platform, Superdome X can achieve industry-leading performance for your particular environment. The tuning parameters suggested this document can be implemented on an as-needed basis. Not all of settings may apply to every application. We recommend testing each suggested optimization for its suitability to your environment

Resources, contacts, or additional links

HP Integrity Superdome X Server
hp.com/servers/superdomex

Simplify Your Linux Experience
hp.com/go/proliantlinux

HP Servers Technology Papers
hp.com/servers/technology

Appendix: FIO profile used in testing

```
$ fio --runtime 30 oracle_profile
```

The contents of the oracle_profile file:

```
; -- start job file --
[global]
bs=8k
ba=8k
direct=1
sync=1
time_based
group_reporting
ramp_time=15

[readers]
rw=randread
numjobs=480
filename=/dev/mapper/mpathb:/dev/mapper/mpathc:/dev/mapper/mpathd:/dev/mapper/mpat
he:/dev/mapper/mpathf:/dev/mapper/mpathg:/dev/mapper/mpathh:/dev/mapper/mpathi:/de
v/mapper/mpathj:/dev/mapper/mpathk:/dev/mapper/mpathl:/o
thinktime=4000
thinktime_spin=200

[parallel_query]
rw=read
numjobs=8
filename=/dev/mapper/mpathb:/dev/mapper/mpathc:/dev/mapper/mpathd:/dev/mapper/mpat
he:/dev/mapper/mpathf:/dev/mapper/mpathg:/dev/mapper/mpathh:/dev/mapper/mpathi:/de
v/mapper/mpathj:/dev/mapper/mpathk:/dev/mapper/mpathl:/o
bs=1024k
ba=1024k
ioengine=libaio
iodepth=4
iodepth_batch=1
iodepth_batch_complete=1
iodepth_low=3
thinktime=40000
thinktime_spin=20000


[writers]
rw=randwrite
numjobs=32
bs=8k
ba=8k
ioengine=libaio
iodepth=400
iodepth_batch=30
iodepth_batch_complete=30
filename=/dev/mapper/mpathb:/dev/mapper/mpathc:/dev/mapper/mpathd:/dev/mapper/mpat
he:/dev/mapper/mpathf:/dev/mapper/mpathg:/dev/mapper/mpathh:/dev/mapper/mpathi:/de
v/mapper/mpathj:/dev/mapper/mpathk:/dev/mapper/mpathl:/o
thinktime=800
thinktime_spin=100
```

```
[redolog]
rw=write
ioengine=libaio
iodepth=6
iodepth_batch=6
iodepth_batch_complete=6
filename=/dev/mapper/mpathp:/dev/mapper/mpathq
thinktime=1000
thinktime_spin=20
sync=1
bs=1k
bsrange=1k-300k
ba=1k
numjobs=1

; -- end job file –
```


4AA5-9698ENW, July 2015