NUMA Best Practices for Dell PowerEdge 12th Generation Servers

Tuning the Linux OS for optimal performance with NUMA systems

John Beckett

Solutions Performance Analysis

Enterprise Solutions Group

Contents

Figures

December 2012 | Version 1.0

Executive summary

Dell™ PowerEdge™ 12th generation servers that use either 2- or 4-processor sockets are NUMA-capable (Non-Uniform Memory Access) by default. Memory on these systems is broken up into "local" and "remote" memory, based on how near the memory is to a specific core executing a thread. Accessing remote memory is generally more costly than local memory from a latency standpoint, and can negatively impact application performance if memory is not allocated local to the core(s) running the workload. Therefore to improve performance, some efforts must be made in Linux environments to ensure that applications are run on specific sets of cores and use the memory closest to them.

With the correct tools and techniques, considerable performance gains can be achieved on memory-intensive applications that may not be completely NUMA-aware. This white paper showcases these tools and gives examples of performance impacts to illustrate how important fine tuning NUMA locality can be in terms of overall performance for some workload types. In addition to the performance uplift of correctly affinitizing applications to specific cores and memory, this paper discusses the concept of NUMA in relation to the PCIe bus, also known as NUMA I/O.

Introduction

Starting in 2003 with the introduction of the 64-bit AMD Opteron™ server processor and followed with introduction of the Intel® Xeon® processor 5500 series, NUMA systems have grown to dominate the x86 server market. The NUMA architecture revolves around the concept that subsets of memory are divided into "local" and "remote" nodes for systems with multiple physical sockets. Because modern processors are much faster than memory, a growing amount of time is spent waiting on memory to feed these processors. Since local memory is faster than remote memory in multi-socket systems, ensuring local memory is accessed rather than remote memory can only be accomplished with a NUMA implementation. The relative locality of a NUMA node depends on which core in a specific processor socket is accessing memory. Accesses to the local NUMA node are normally lower latency and higher bandwidth than memory accesses to remote NUMA node(s).

Linux NUMA awareness has improved dramatically from the early days of adoption in the 2.4 kernel tree, and the 2.6 and 3.0 kernels providing increasingly robust NUMA capabilities. Today, schedulers with the latest Linux kernels, device drivers, and many applications have substantial NUMA awareness, and it is increasingly rare to encounter applications to suffer significant performance penalties without explicit affinity direction from the user. However corner cases still exist and this white paper focuses on the toolset needed to analyze the NUMA topology of a given Dell server and manually apply affinity to achieve optimal workload performance.

Test methodology for non-I/O NUMA characterization

For the purposes of this white paper, we measured NUMA characteristics in a variety of ways. To examine the performance characteristics of the memory subsystem itself, memory bandwidth and memory latency were measured. Memory bandwidth represents the rate at which memory can be read from or written to by a processor. Memory latency is the time it takes to initiate a 64-byte message transfer. Both metrics are important for evaluating memory subsystem performance.

To measure the impact of NUMA on actual workloads, a set of processor intensive benchmarks were chosen to characterize binding impacts on performance. SPEC CPU2006 was primarily chosen for this role, due to the fact that the benchmark is comprised of fully-multithreaded integer and floating point workloads that show a variety of impacts of NUMA localization.

Memory bandwidth was measured using a version of the common STREAM benchmark, which is the industry standard tool for measuring memory bandwidth. The version used for this test was compiled with OpenMP (parallel) support and optimized for the new generation of Intel Xeon processors. When testing overall system memory bandwidth, optimal results are achieved by running the STREAM benchmark with one thread per physical core. Although STREAM uses four separate vector kernels during the course of measurement (Add, Scale, Copy, and Triad), the Triad value is used for the purposes of this white paper. STREAM results are returned in values of MB/sec.

Latency measurements were taken using the lat_mem_rd subtest, part of the LMBENCH benchmark. Lat_mem_rd measures memory-read latency for varying memory sizes and strides. The results are reported in nanoseconds per load. The entire memory hierarchy is measured, including onboard cache latency and size, external cache latency and size, and main memory latency. Both local and remote NUMA node main memory latency is explored with lat_mem_rd, localizing the process and forcing memory access to local or remote using the numactl tool in Linux. The last 20 lines of output from main memory measurements of lat_mem_rd are averaged, and each test executed for three iterations. The median average result for each test type was selected to report latency results for this white paper.

In addition, the concept of “loaded latency” is explored for some configurations. This is intended to represent average memory latencies when the NUMA node is already under heavy load. The workload chosen to provide the heavy memory load is STREAM, and n-1 real cores on a single socket were tasked with separate single-threaded STREAM processes to load the local NUMA node targeted for latency experiments. The single free core on the physical processor under test was tasked to run the latency measurement, measuring memory latency to the local NUMA node. As with other latency measurements, the last 20 lines of output were averaged to produce the final loaded latency value. Each loaded latency test was run three times, and the median loaded latency average was used for the official result.

SPECrate metrics from SPEC CPU2006 (hereafter referred to as SPECint_rate_base2006 and SPECfp_rate_base2006) were also chosen for performance characterization, as they are the leading industry standard CPU-centric benchmarks that use all available processor cores and threads. Each benchmark is comprised of multiple sub-benchmarks, each tailored to model different common computing tasks or scientific workloads, broken out into either integer or floating point. Both int_rate and fp_rate are throughput-based benchmarks. The “base” portion of the benchmark name indicates that a standardized set of compiler options were used to create the benchmark binaries. The SPECfp_rate_base2006 benchmark in particular can be used as a proxy for scientific workloads, and tends to be more sensitive to memory frequency and bandwidth differences than SPECint_rate_base2006. All CPU2006 measurements were conducted with the Red Hat® Enterprise Linux® (RHEL) 6.2 operating system. The effects of explicit thread binding to cores per NUMA nodes were measured.

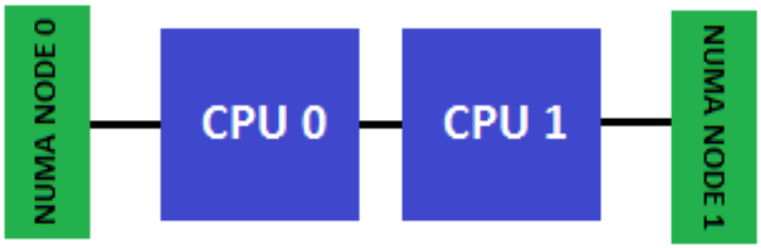
NUMA I/O was explored using a specialized set of tools; see the NUMA I/O section on pg. 26 for details.

Introduction to NUMA topology

When assessing a NUMA system for the first time, it is important to be able to visualize the NUMA topology. One way to think about this is to picture the physical processors in relation to the populated memory. For systems that have more than one physical processor (NUMA capable systems, the focus of this white paper), memory can be either remote or local.

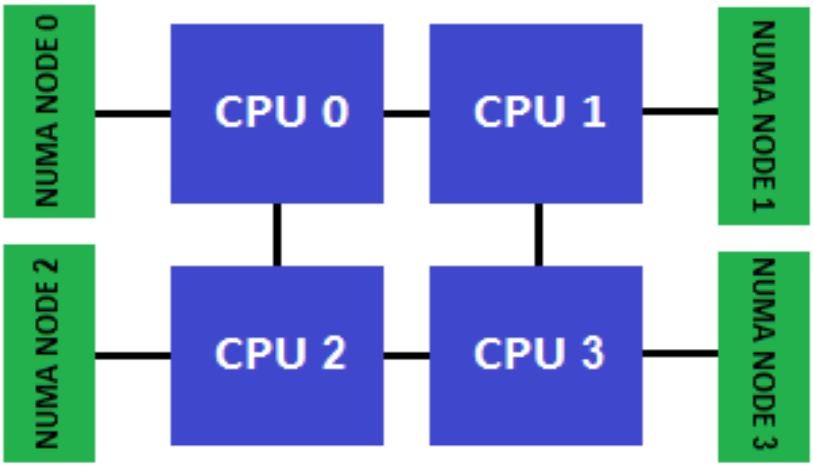
Figure 1 illustrates the highest level 2-socket Intel NUMA topology. In this example, CPU 0 and CPU 1 represent the physical processor package, not individual cores, and a corresponding number of NUMA memory nodes assuming memory modules are populated in DIMM slots adjacent to each processor.

Figure 1. 2-socket Intel NUMA node architecture



For Intel Xeon E5-4600-based systems such as the PowerEdge R820 and M820, the highest level NUMA topology becomes more complex due to the nature of the 4-socket design. For these systems, if there are four populated processor sockets, there are a corresponding number of NUMA nodes. Figure 2 illustrates this example.

Figure 2. 4-socket Intel E5-4600 NUMA node architecture



For 4-socket processor AMD Opteron-based systems, such as the PowerEdge R815 and M915, the NUMA layout becomes considerably more complex due to the fact that each physical processor package has two NUMA nodes. For these systems, if there are four populated processor sockets, there are eight NUMA nodes. Refer to Figure 3 for 4P AMD example topology.

Figure 3. 4-socket AMD NUMA node architecture

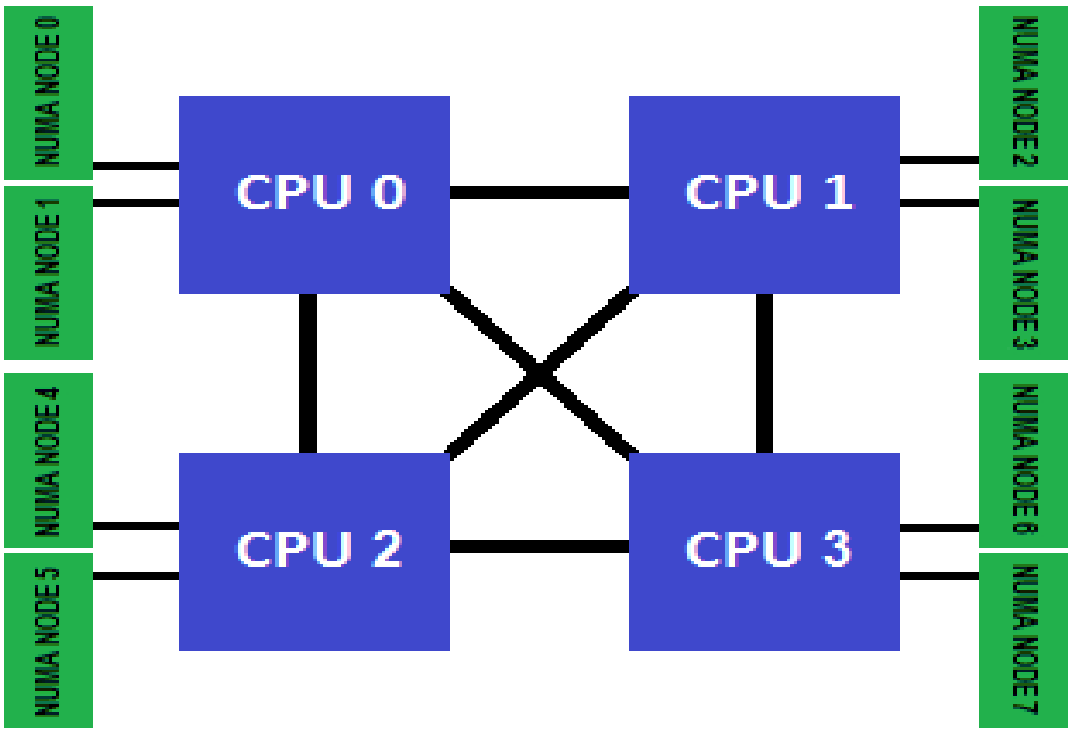


Figure 4 illustrates an introduction to NUMA node locality (in which NUMA node is considered local to a given core). In this example, for core 0 in physical processor CPU 0, the memory immediately adjacent to that processor socket is considered the local NUMA node. For core 1, which is part of physical processor CPU 1, the NUMA node considered local is the one hanging off of CPU 1. Each physical processor can have up to eight physical cores with the Intel Xeon E5-2600 family of processors, and up to 16 total logical processors (with Hyper-Threading enabled) per socket. For the purposes of illustration, only the first physical core on each processor socket is shown.

Figure 4. NUMA locality relative to individual cores

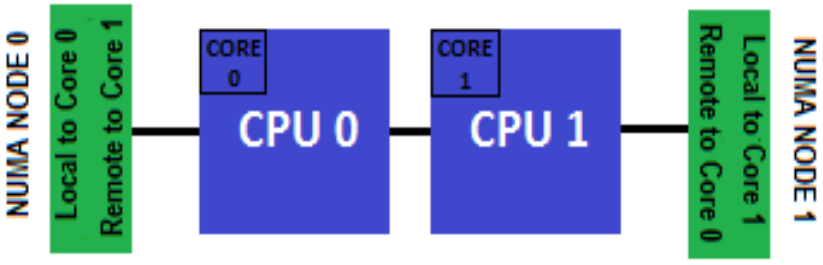


Figure 5 represents another way to visualize relative NUMA node locality. Here, we are looking at another simplified example where each example top-bin E5-2600 series physical processor in a socket (represented as CPU 0 and CPU 1) has eight cores each. For the purposes of this illustration, each processor core is numbered from one through eight although that core numbering strategy changes post-boot. We can see that the first core on CPU 0, colored in green, is local to NUMA node 0. This means that the DIMM slots populated closest to CPU 0 are local, and the DIMM slots populated closest to CPU 1 (NUMA node 1 in red) are remote. This is due to the fact that to reach NUMA node 1 from Core 1 on CPU 0, the memory request must traverse the inter-CPU QPI link and use CPU 1's memory controller to access this remote node. The addition of "extra hops" incurs latency penalties on remote NUMA node memory access.

Figure 5. NUMA node/core locality example – node 0

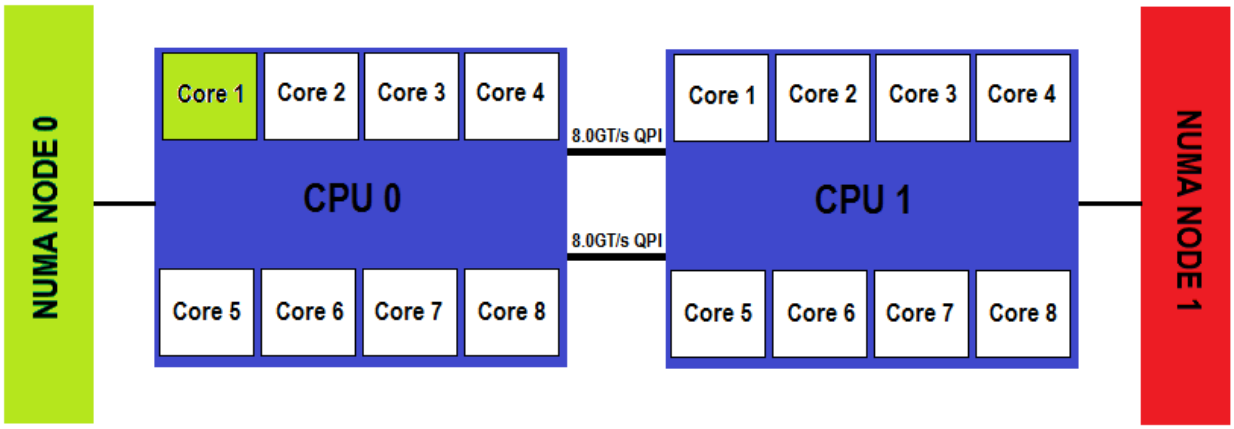
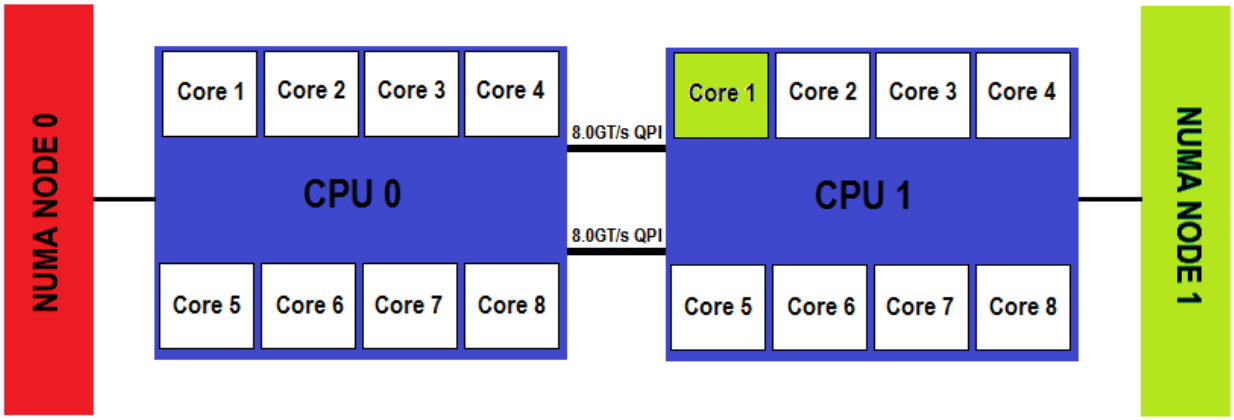


Figure 6 shows another similar example, but this time we are looking at the first core on the second physical processor (CPU 1). Again, for the purposes of this example, we are looking at a simplified numbering structure of cores where each physical processor has eight cores, numbered one through eight. This illustration shows that for the first core in the second physical processor (colored in green); the NUMA node considered local to it is NUMA node 1. For this specific core, any memory accesses to NUMA node 0 are considered remote and will incur some latency penalties. These examples are building blocks in our understanding of NUMA topology, but we must examine the concept of core enumeration to get a fuller understanding of how to apply these concepts on a running Linux system.

Figure 6. NUMA node/core locality example – node 1

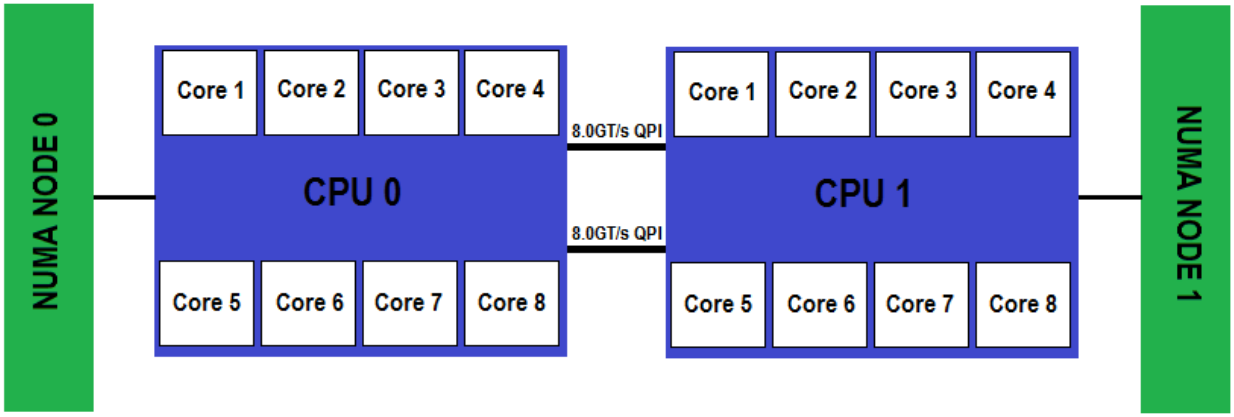


Core enumeration in Linux

As useful as the examples above are for visualizing basic NUMA topology, it is critical to understand how to translate this understanding to being able to determine which logical processor in a running Linux operating system is on a given socket or local to any particular NUMA node. When the Linux kernel is booting, processor cores on each physical processor package are translated to operating system CPUs (OS CPUs) with a particular order dictated by the BIOS ACPI table. **It is crucial to understand that different vendor server platforms will enumerate cores in Linux differently due to different ACPI presentations. Each server platform should be evaluated for core enumeration separately, especially if adopting tuning strategies originally developed on another vendor's system.** All Dell PowerEdge Intel Xeon-based platforms enumerate cores in the same manner.

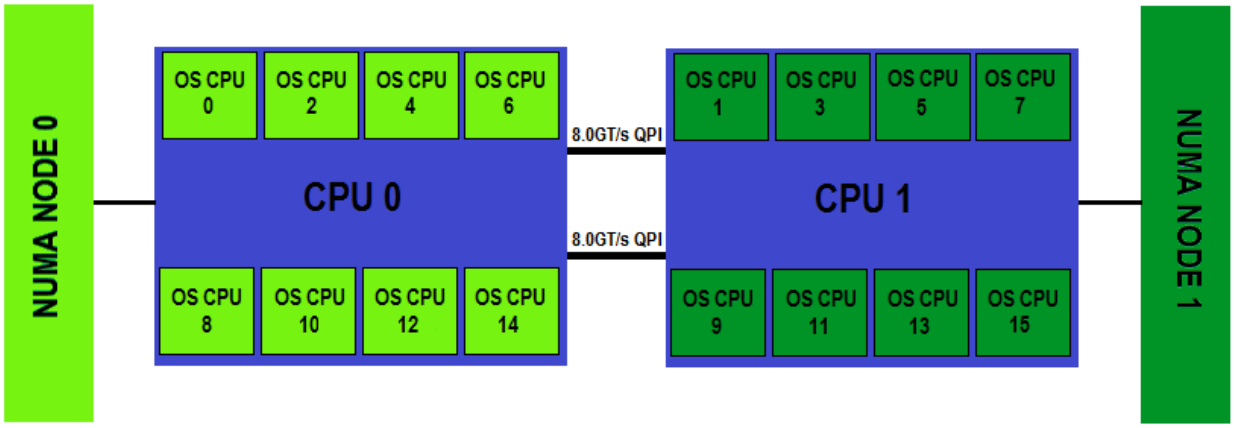
In Figure 7, we again see a basic representation of a 2-socket, 16-core Intel Xeon-based system, where each physical processor package contains eight processor cores. **For the purposes of this illustration, each processor core is numbered from one through eight although that core numbering strategy changes post-boot.** Although we have seen an illustration similar to this, the progression of high to low level topology illustrations will help the reader successfully visualize their own system's NUMA topology and make informed decisions on choosing manual affinity strategies.

Figure 7. High-level 2P Intel core ordering (pre-Linux boot)



In Figure 8, we see the same system represented, but now Linux has enumerated the cores. Note that each core is now represented as a different "OS CPU", which is how Linux refers to logical processors. It is also important to note that the first OS CPU (0) is the first core on the first physical processor package. Now note that OS CPU 1 is the first core on the second physical processor package. The pattern continues across all available cores, so **for an Intel 2P system, all even numbered OS CPUs are on the first physical processor package, and all odd numbered OS CPUs are on the second physical processor package**. It is also important to note which cores are local to any particular NUMA node, and that OS CPUs and NUMA nodes are numbered starting from 0.

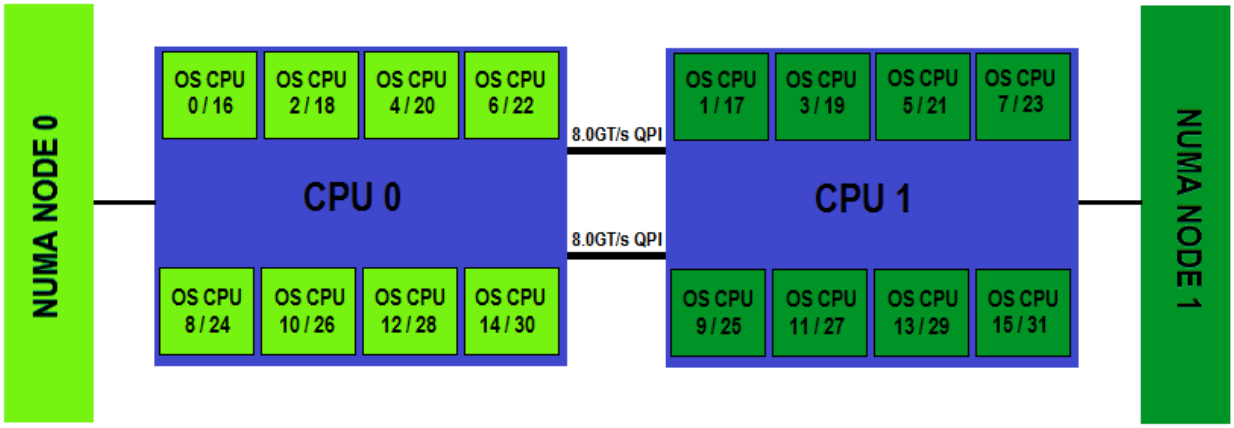
Figure 8. 2P Intel Linux core enumeration – Hyper-Threading disabled



For Intel Xeon-based systems, there can be up to two logical processors per physical core when Hyper-Threading is enabled. The two logical processors represent one “real core” and one “Hyper-Threaded sibling”. When Hyper-Threading is disabled in BIOS Setup (typically represented as the Logical Processor option), only one logical processor will be presented to the operating system per physical core (as illustrated in Figure 8). The Logical Processor option is enabled by default on Dell server platforms.

The core enumeration pattern represented in Figure 9 shows the same 2-socket processor Intel Xeon-based system with Hyper-Threading enabled. The first number before the “/” character represents the OS CPU number assigned to the real core. The second number after the “/” character represents the OS CPU number assigned to the Hyper-Threaded sibling. During the core enumeration phase of the Linux kernel boot, all real cores are enumerated first in a round-robin fashion between populated physical processor packages. Once all real cores are enumerated, the Hyper-Threaded siblings are enumerated in a similar fashion, which round-robin between populated processor packages. For the example of a 2P 16-core system with Intel Xeon 8-core processors, 32 total logical processors are enumerated when Hyper-Threading is enabled.

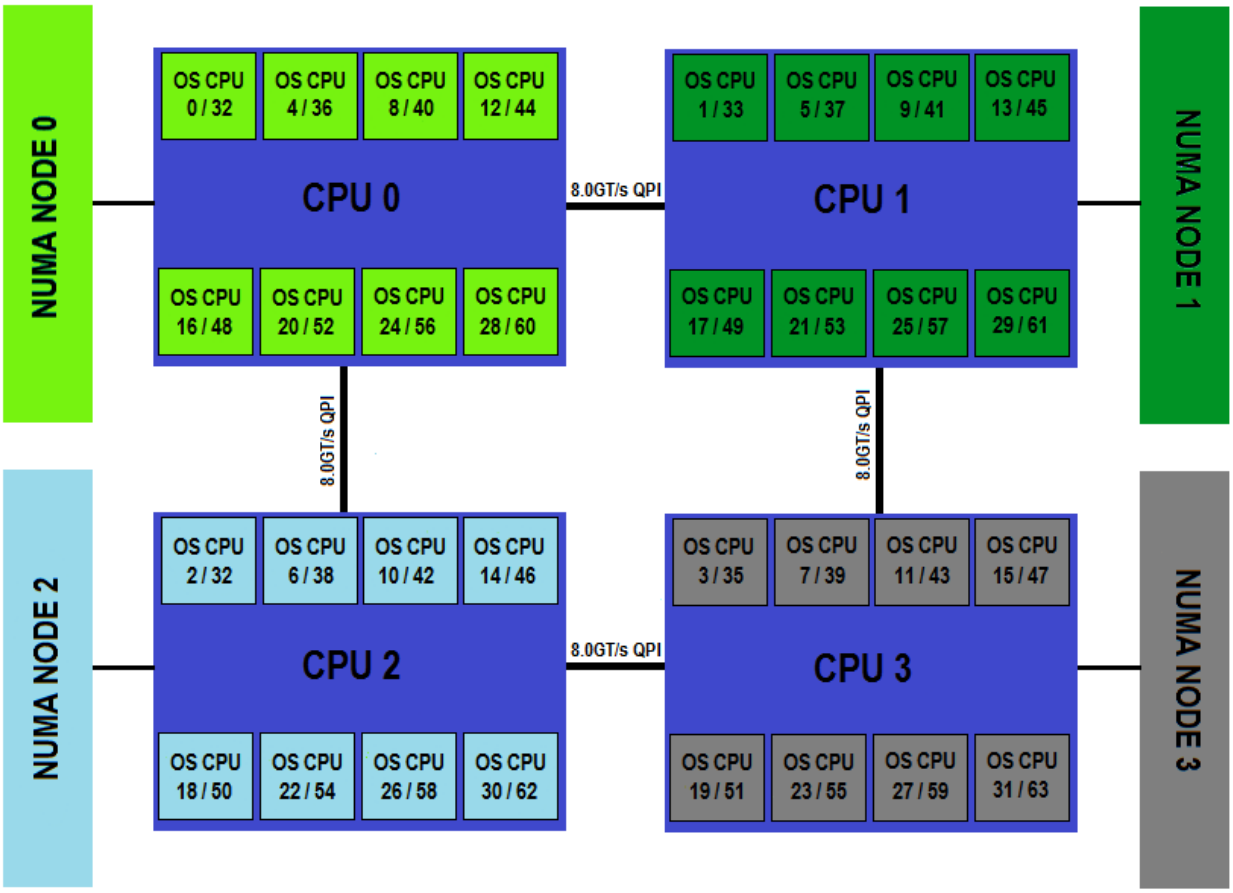
Figure 9. 2P Intel Linux core enumeration – Hyper-Threading enabled



For PowerEdge systems such as the R820 and M820 that support the Intel Xeon E5-4600 family of processors, systems can have up to four physical processors populated. If we examine the core enumeration of a 4P system with 32 total cores, where each physical processor package contains 8 cores we find that 64 total logical processors are present in the Operating System when Hyper-Threading is enabled. In Figure 10, the core enumeration is shown for this platform. Note that real cores are again enumerated first in a round robin fashion between physical processor packages. After all of the real cores are enabled, the Hyper-Threaded siblings are enabled in the same round robin pattern.

Because of the fact that a given core can be up to two QPI hops away from one of the four NUMA nodes based on the "box" topology, it can be more important to aggressively affinitize workloads for this architecture. Users are encouraged to experiment with manual affinity for these platforms for optimal performance.

Figure 10. 4P Intel Linux core enumeration – Hyper-Threading enabled

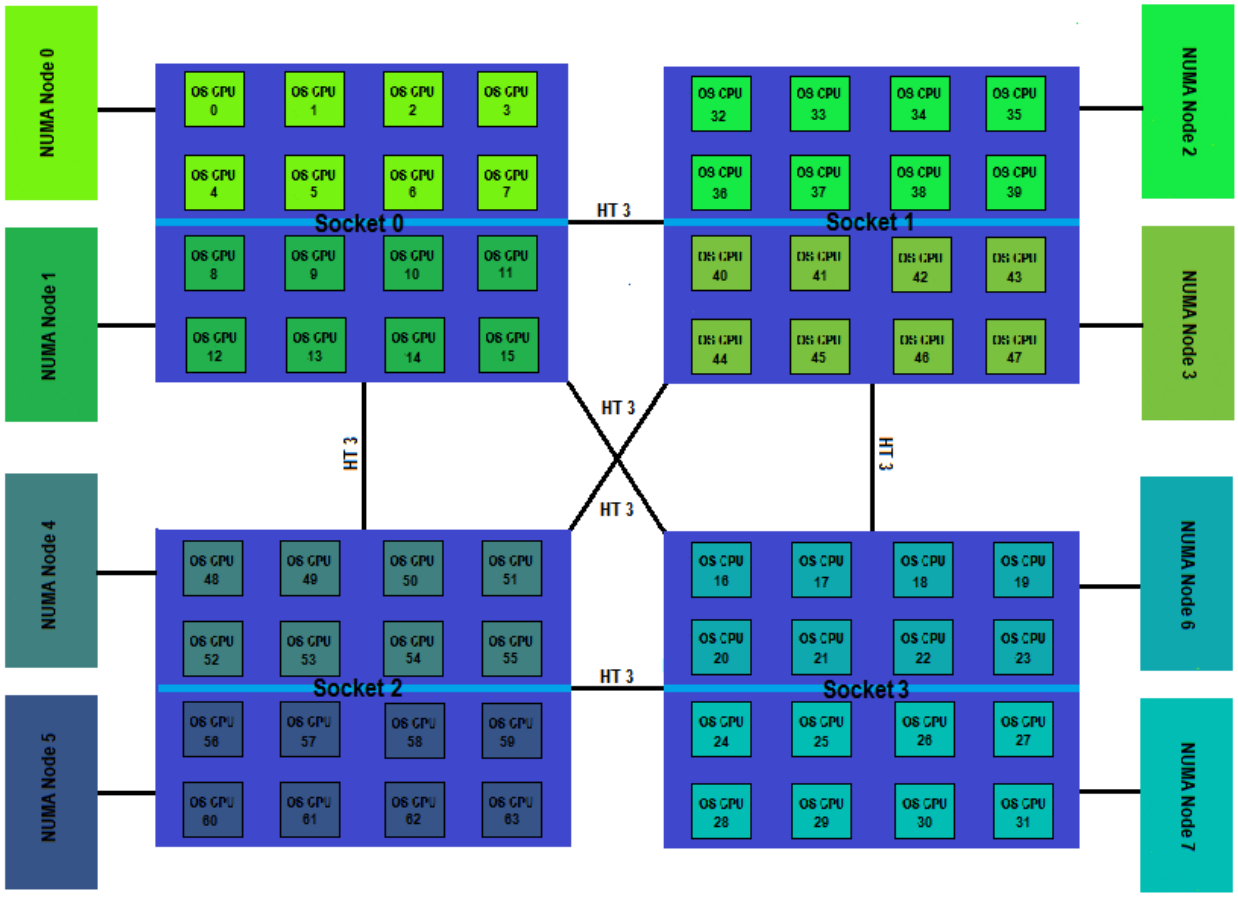


AMD Opteron-based systems do not utilize the concept of Hyper-Threading. Instead, with the introduction of the Interlagos processor, pairs of real cores retain independent integer schedulers and pipelines, but share floating point schedulers and pipelines. Figure 11 shows the core enumeration of an AMD Opteron 6400-series (Abu Dhabi) 4P system with 64 total cores. As we can see in the illustration, the core enumeration strategy for this system is drastically different than Intel platforms.

Note that each physical processor package has two NUMA nodes associated with it. In addition, each physical processor is broken up into two sets of 8 cores each. The cores associated with a single NUMA node are enumerated in order. In Figure 11 we see OS CPU 0 – 8 are part of NUMA node 0. The second set of cores on the same physical processor package numbered OS CPU 9 – 15 are local to NUMA node 1. It is critical to note that shifting to the next socket has the cores enumerated starting with 32. The fourth physical processor package starts core enumeration with OS CPU 16 and carries onward.

This representation depicts a 4P PowerEdge R815, but it is important to examine each AMD-based Dell platform for core enumeration strategy, as subtle differences do exist between Dell PowerEdge AMD Opteron-based systems dependent upon how the HyperTransport links are laid out on the system board.

Figure 11. 4P AMD Abu-Dhabi Linux core enumeration

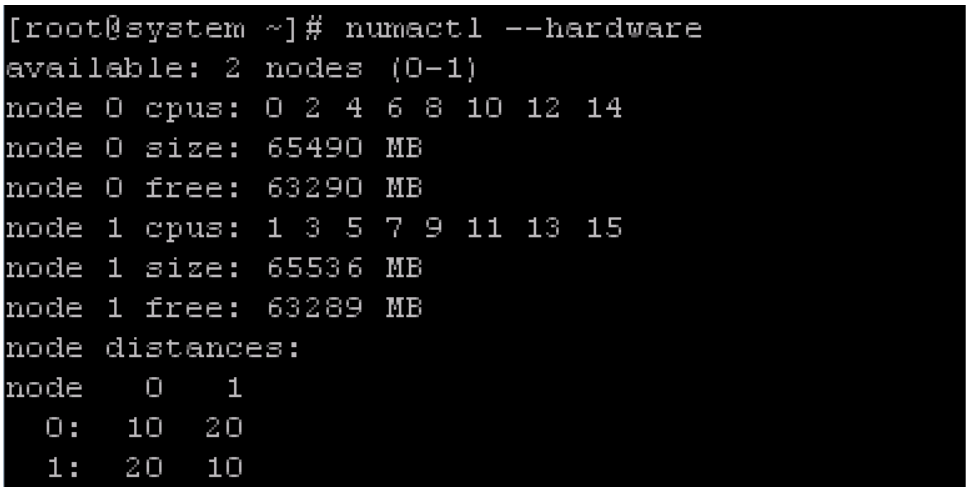


Examining NUMA topology in Linux

There are several commands typically available in more modern server-oriented 64-bit Linux distributions, such as Red Hat Enterprise Linux (RHEL) and Novell® SuSE® Linux Enterprise Server (SLES) that provide tools for examining a platform's NUMA topology. These tools include the following commands:

- **numactl --hardware:** This command shows the NUMA node count, the individual logical processors associated with each node, and the node distances. This command is greatly useful for determining which cores are local to an individual NUMA node for later use in process affinitization. Figure 12 shows the output of this command on an example E5-2600 series dual-processor 16- total core system with 8 real cores per physical processor package. Hyper-Threading is disabled, and so 16 logical processors are represented.
  - The **available** line indicates the total number of NUMA nodes, in this case 2. They are numbered 0 and 1.
  - **Node 0 cpus**: This line describes the logical processors that are "local" to NUMA (memory) node 0. This means that all of these cores and Hyper-Threaded siblings are on a single physical processor. Remember, this list includes both physical cores and Hyper-Threaded siblings, so the count is double to the number of physical cores in a given processor.
  - **Node 0 size**: This is the total amount of memory contained in this NUMA node. Since the example 2P Intel Xeon system has a memory configuration of 16 x 8GB DIMMs, 8 DIMMs are populated per CPU. The node size of 32690MB correlates to the fact that 8 DIMMs x 8GB = 64GB.
  - **Node 0 free**: This is the amount of currently unallocated memory in NUMA node 0. This is important to keep in mind when deciding which process should go where when evaluating manual affinitization options. In some circumstances, it can be helpful to alternate different bindings between nodes for different separate processes such that you don't exhaust the memory in one NUMA node while leaving the other untouched.
  - **Node 1 lines**: Refer to the above statements for general understanding. Node 1 information describes cores, memory, and so on, and is local to the second physical processor in the example system.
  - **Node Distances**: This is a table that shows the relative distances between NUMA nodes. The distances represented can be classified as near (10), remote (20), and very remote (20+). The distances are provided through a BIOS provision, and the values are dependent on CPU topology and inter-CPU link constraints.

Figure 12. Numactl –hardware example (Hyper-Threading disabled)



- **numastat:** This command displays NUMA allocations statistics from the kernel memory allocator. Each process has NUMA policies that specifies which node pages are allocated. The numastat counters keep track on what nodes memory is finally allocated. Figure 13 shows the output of the numastat command on a system that has been running for some time and has had several benchmarks run that were explicitly affinitized to one NUMA node or the other.

  The counters are separated for each node. Each count event is the allocation of a page of memory.

  - **numa_hit** is the number of allocations which were intended for that node and succeeded there.
  - **numa_miss** shows the count of allocations that were intended for this node, but ended up on another node due to memory constraints.
  - **numa_foreign** is the number of allocations that were intended for another node, but ended up on this node. Each numa_foreign event has a numa_miss on another node.
  - **interleave_hit** is the count of interleave policy allocations which were intended for a specific node and were successful.
  - **local_node** is a value that is incremented when a process running on the node allocated memory on the same node.
  - **other_node** is incremented when a process running on another node allocated memory on that node.

Figure 13. Numastat example

```
[root@system ~]# numastat
                           node0           node1
numa_hit                 472237          244584
numa_miss                     0               0
numa_foreign                  0               0
interleave_hit            32074           32081
local_node               471546          206802
other_node                  691           37782
[root@RHEL-latency1 ~]#
```

- **numactl --show**: This command displays the current memory allocation policy and preferred NUMA node (see Memory allocation scopes and policies), as well as a list of valid OS CPUs and node numbers for other numactl flags such as --physcpubind, --cpubind, and --membind.

Figure 14. Numactl --show example

```
[root@system ~] numactl --show
policy: default
preferred node: current
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
 26 27 28 29 30 31
cpubind: 0 1
nodebind: 0 1
membind: 0 1
```

Linux Resources for NUMA management

There are a growing number of resources available in Linux to examine NUMA memory allocation, change default behaviors, and force specific logical processor and memory affinity upon new and running applications. Linux allows for several memory allocation policies that can affect memory allocation behavior. The default choices for memory management can have potential drawbacks in heavily loaded environments or particular computing scenarios. To get a complete understanding of underlying mechanisms which can impact how memory pages are allocated in a running environment, general Linux memory management should be understood. Due to the complexity of this topic, many low level details are not discussed in this whitepaper. For a deeper understanding of memory management concepts in Linux Operating Systems, please read this primer.

Memory allocation scopes and policies

For recent NUMA-aware Linux distributions, a set of configurable memory policies allow the user to change memory allocation strategies. There are several components of a Memory Allocation Policy: scope, mode, mode flags and specific nodes provided as an argument. The scope governs the inheritance patterns and may limit the types of memory regions certain policy modes may operate on. Memory Allocation Policy mode is the specific strategy to allocate memory, and is controlled by optional flags and a node list. Many of the available Memory Allocation Policy scopes and modes are primarily utilized by specific application design techniques to enable NUMA-awareness. However, some elements are useful for user-level memory allocation tuning, especially for non-NUMA aware applications. As such, only the policies of particular interest for user-level tuning here. The use of shared memory policies and scopes are of potential interest for user-level tuning under some circumstances, but are not in the scope of this whitepaper. Detailed information on Linux Memory Allocation Policies, scopes and modes can be found here.

Memory policy scopes

- The **System Default Policy** is called **local allocation**. This policy dictates memory allocation for tasks that take place on the NUMA node closest to the logical processor where the task is executing. This policy is a reasonable balance for many general computing environments, as it provides good overall support for NUMA memory topologies without the need for manual user intervention. However, under a heavy load, the scheduler may move the task to other cores and leave the memory allocated to a NUMA node now remote from where the task is executing. Tools to prevent the scheduler from migrating workloads are described in the NUMA affinity tools in Linux section. The scope of this default memory allocation policy is set system-wide, and all tasks will inherit this behavior unless set by a NUMA-aware application or by manual user intervention.
- The **Task/Process Policy** is the policy scope is selected when defined for a specific task, generally through manual user intervention. This policy scope applies to the task's entire address space, and is inherited across fork() and exec() calls. This scope is particularly useful for user-level tuning, as calls to the numactl command for binding purposes can use this allocation policy. The Task/Process policy scope is utilized when explicitly affinitizing a shell scripts or executables that spawn applications.

Memory policy modes

- **Bind mode**: This policy mode requires that all memory allocation must come from the supplied NUMA node numbers supplied as arguments. This policy does not allow for memory allocation outside of the specified NUMA nodes, so there are hard limits to its use. The use of numactl with the --membind or --localalloc option use the Bind mode.
- **Preferred mode**: This policy mode requires that memory allocated will first be directed to the specified single node supplied as an argument. If this node is full and memory allocation fails, the memory will be allocated to adjacent nodes based on node distance.

- **Interleave mode**: This memory policy mode forces that each page allocation is interleaved between a set of node numbers specified as an argument. This mode is useful in scenarios where there is not enough available logical processors local to a node, or there is not enough memory in one NUMA node for a large workload to obtain sufficient resources.

Other useful Linux options and tools

- **zone_reclaim_mode**: Another helpful Linux feature is the ability to help Linux retain memory pages on local NUMA nodes by aggressively freeing up pages on the Node under contention. This is primarily useful where a particular NUMA node is in danger of being over-subscribed, and subsequent memory allocation is in danger of being forced onto an alternate node. The location of the kernel tunable is **/proc/sys/vm/zone_reclaim_mode**. Echoing a value of 1 to this file will enable aggressive page reclamation. More information regarding this feature can be found here.
- **numa_maps**: Another interesting Linux feature that exposes information about NUMA memory allocation for a running process is the /proc/<process id>/maps file. The kernel keeps track of memory allocation information for each process running on the system. Much information can be gleaned from these files, but one of the more interesting elements from a NUMA localization perspective is the ability to examine on which NUMA node particular memory pages have been allocated for the constituent objects. In addition, the memory policy of a process can be obtained by analyzing this file. Care should be used when operating on this file, as it is generally not recommended to query processes in a production environment, especially at a high rate. More information on this valuable analytic tool can be found here.
- **Cpusets and Cgroups**: Processors and memory may also be allocated manually into sets of resource groups known as cpusets or cgroups, depending on the interface used. The usage of these features falls outside of the scope of this white paper, but much additional control of system resources can be leveraged by the use of this feature and its usage in confining specific application processor and memory utilization to a selected subset of available processors and memory. Running processes can be moved between cpusets, and memory migration and allocation can be fine-tuned with the use of this feature. More details can be found here and here.
- **The numad Daemeon**: New for RHEL6.3, this daemon is intended to discover the system's NUMA topology and monitor processes that utilize significant processor and memory resources. The daemon can provide automatic NUMA alignment for such processes, and additionally will provide access to NUMA advice via command line or an API to query for pre-placement. This daemon holds considerable promise to lighten the burden of manual affinitization in the future, but is too new to have been characterized fully.

NUMA affinity tools in Linux

Linux has several tools that allow you to explicitly affinitize a process to particular cores and potentially localize memory allocation to specific NUMA nodes. Before the advent of common NUMA aware systems, commands such as **taskset** were used to bind a process or command to a particular core. The taskset command is not NUMA-aware, so memory allocations are not locked to any particular NUMA node. However, the default memory policy in recent Linux distributions will ensure initial allocation to node(s) local to the logical processors specified in the affinity mask. Taskset has lost some of its usefulness in a NUMA environment, but still has some capability in allowing non-NUMA-aware processes or command binding. One of the more useful features of taskset is the capability to show the affinity mask of a process already running.

Useful features of taskset

To determine the OS CPU affinity mask for a running process, use:

taskset -p <process id>

This command returns a hex mask that you can use to determine which OS CPUs are bound. If the mask returned is all Fs, this means no explicit affinity has been provided and the scheduler is free to move the process around to any available logical processor.

You can also use this taskset command to change the affinity mask of a running process, which will also enforce new pages allocated to the process to be mapped to the new local NUMA node (based on the CPU mask provided). However, previously allocated memory is not automatically migrated.

taskset –p <cpu mask> <process id>

Taskset can also be used to start an executable with non-NUMA-aware binding to one or more specified logical processors.

taskset <cpu mask> <application executable>

The flexible numactl command

In order to provide for optimal performance on a NUMA aware system, it can be important to pin performance-sensitive workload threads to specific cores local to one physical processor, and ensure that all memory allocation for that process occurs on the local NUMA node. This provides for two potential performance enhancements. By executing a particular workload on specific OS CPUs, you can prevent the operating system scheduler from shuffling processes around to different cores during execution, as can occur under heavy load. By fixing the cores for process execution, you can target multiple threads on processors that share the same L3 cache. The workload can also be affinitized to a particular NUMA node to maximize local memory access.

Using the **numactl --hardware** command, we see the following pertinent information in Figure 15 from a 2P Intel platform with 16 real cores, Hyper-Threading enabled, 32 total logical processors. Note that under a running system with some workloads already running, some NUMA nodes may not have enough memory available to allocate to a new application. It is important to scrutinize the **node free** lines to ensure sufficient free memory is available for allocation. If a likely NUMA node has been located with enough free memory, examine the **node cores** for the identified NUMA node for subsequent use**.**

Figure 15. Numactl --hardware example (Hyper-Threading enabled)

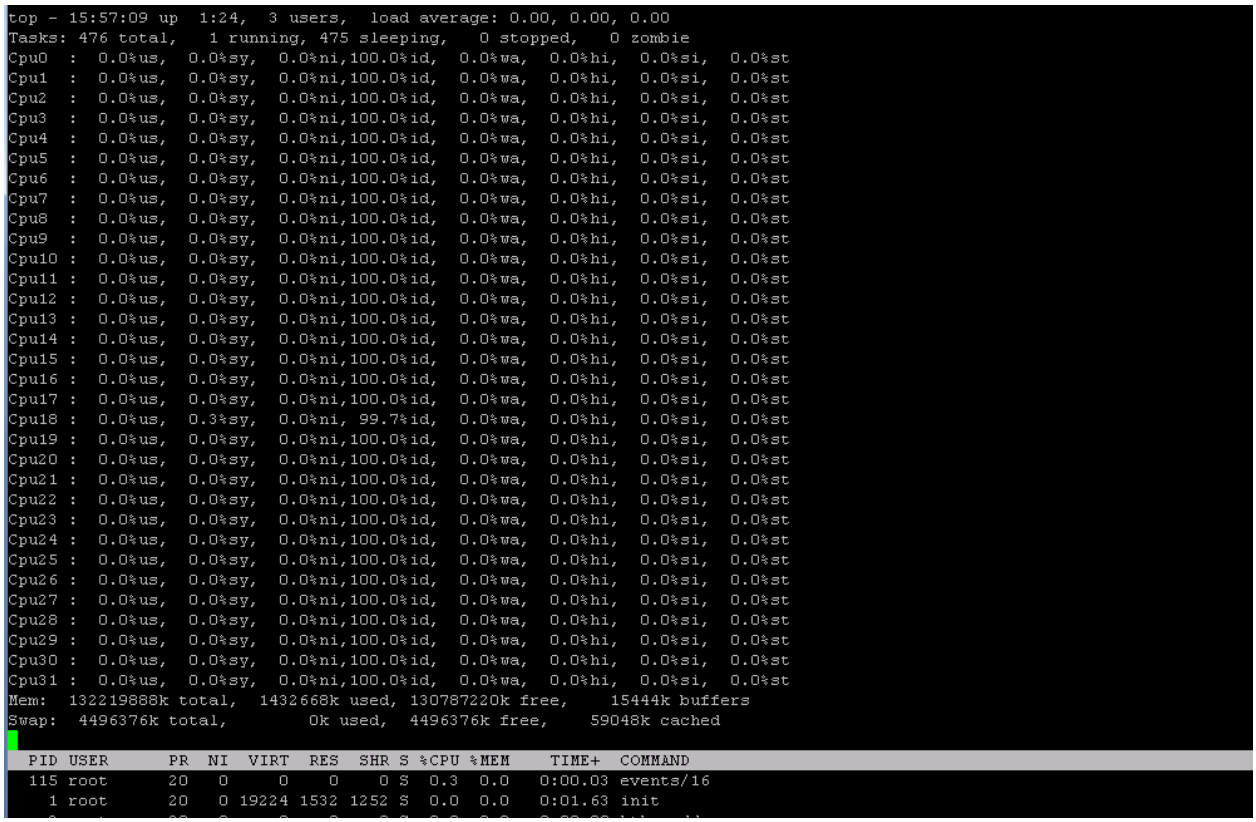
```
[root@system ~] numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
node 0 size: 65458 MB
node 0 free: 63752 MB
node 1 cpus: 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31
node 1 size: 65536 MB
node 1 free: 63976 MB
node distances:
node   0   1
  0:  10  20
  1:  20  10
```

Figure 16 shows the output of the “top” command, with a subsequent key press of the “1” key. This will display current usage statistics for all available logical processors instead of the default averaged processor utilization view. The top command samples the processors every few seconds and updates the display continuously until exited by pressing the “Q” key.

When attempting to locate free cores to bind a new workload to, make sure OS CPUs of interest are not already heavily tasked with other work. This is accomplished by noting the idle level for specific cores by looking for the **xx.x% id** column and matching this to the corresponding OS CPU row. Figure 16 shows a system almost completely idle, with all but one logical processor showing 100% idle.

The top command has many other uses than this example, and is a valuable tool for performance diagnostics. Please consult the top man page for a great variety of additional features that can be leveraged.

Figure 16. Example of top command output



Binding a workload using numactl

The command to bind a single threaded workload to a single specified logical processor and to local memory is as follows:

numactl -l --physcpubind=2 <*application executable*>

The "-l" flag tells Numactl to allocate only to local memory, which is dependent upon the logical processors that the application starts on.

The "--physcpubind=2" flag in this example tells numactl to bind this application only to OS CPU 2.

For a multi-threaded application example running with 4 threads, assuming OS CPUs 2,4,6 and 8 are on the same NUMA Node:

numactl -l --physcpubind=2,4,6,8 <*application executable*>

Or, assuming there are at least four logical processors available on the same NUMA node, this command will bind the application to cores available on a single NUMA node and ensure local memory allocation:

numactl -l --cpunodebind=0 <*application executable*>

Note that all the specified OS CPUs are on the same physical processor, and local to the same memory node. If you specified “2,3,4,5” for the --physcpubind flag, this would likely be suboptimal as these are a mix of OS CPUs from two separate NUMA nodes, and will incur caching and memory latency penalties. Note that using the -l flag in conjunction with a list of logical processors specified with the --physcpubind flag that are not on the same NUMA node normally causes an error.

In cases where an application must be affinitized to a set of cores that are not all on the same NUMA node due to existing resource constraints, the memory may be allocated to more than one NUMA node with the -m <nodes> flag, where <nodes> is a comma separated numeric node list. In cases such as this, it may be advisable to evaluate whether the --interleave <nodes> flag may produce superior results, since this ensures even distribution of memory allocation to the specified NUMA nodes.

It is sometimes advisable to leave at least one core on the system free for the OS kernel to run in situations where application affinitization has targeted most of the available logical processors. Commonly the choice is OS CPU 0, but this is not mandated. Some customers may benefit from keeping one core per physical processor free from explicitly affinitized applications (usually the first OS CPU in each NUMA node), but it depends on the environment and the nature of the workload(s).

Some other valuable flags available for the numactl command include the following.

--membind=<Node> or -m <NUMA Node>

Use this option to specify the numbered NUMA node to force memory allocation when starting a new process. More than one NUMA node may be specified, but will not be optimal unless facing free memory constraints.

--physcpubind=<OS CPU list> or –C <OS CPU list>

This option forces the workload to be bound to a particular OS CPU or set of OS CPUs. To specify more than one OS CPU, the list can be specified as a range (1-4), or specified as a comma-separated list. The latter is preferred for Dell systems as it is not common for a NUMA system to have contiguous OS CPUs enumerated on the same core.

--cpunodebind=<NUMA Node> or -N <NUMA Node>

Use this option to specify that the process starting be allocated to any logical processor local to the named NUMA node. More than one NUMA node may be specified, but a single node is typically optimal for performance if enough free logical processors exist associated with a NUMA node.

--interleave=<NUMA Nodes>

The interleave option is useful for distributing an application's memory allocation across multiple NUMA nodes. Scenarios where this might be useful would be for applications whose total memory requirements are larger than a total memory in a single NUMA node. Other situations envisioned might be for 4P systems where two of the four processor sockets are local to targeted I/O devices, and application performance may be advantaged by keeping memory allocated to Nodes closest to the PCIe devices. By interleaving memory allocation between NUMA nodes, memory latency and bandwidth should be averaged between the local and remote node(s) identified for memory allocation.

Manually migrating memory pages between NUMA nodes

It is possible under some environments that it may be beneficial to move memory allocation from one NUMA node to another for a running application. The **migratepages** command contained in the numactl package allows the user to force all allocated memory pages associated with a particular process id from one or more NUMA nodes to other NUMA Node(s). This can be useful on a running system to manually balance memory usage, but does carry risks of disrupting target NUMA node memory allocation if the pages to be moved from the source node are larger than the available free space in the target node. More detail can be found in the migratepages() man page.

The form of the command is as follows:

migratepages <process id> <current NUMA Node list> <target NUMA Node list>

NUMA affinity performance measurements

The following section demonstrates measured performance deltas between local and remote NUMA nodes. The importance of proper NUMA affinity cannot be effectively defined without data showing best and worst case scenarios. Although the default memory management policy will attempt to allocate memory local to the logical processors executing the workload, under many scenarios the memory may end up running remotely due to load balancing between available logical processors. Without explicit NUMA affinity, the end user may still experience remote memory allocation. In order to quantify these performance deltas, memory bandwidth and latency are characterized from a local and remote NUMA allocation perspective. The effects on fully-multithreaded workloads such as SPEC CPU2006 int_rate and fp_rate are used as a proxy for the varying NUMA allocation effects that can be expected when running disparate workload types.

Measuring memory bandwidth and latency across NUMA nodes

In Figure 17, we see that in terms of memory bandwidth, there is an approximately 50% decline going to a remote NUMA node compared to the local NUMA node. For the purposes of this illustration, cores initiating the STREAM test were on socket 0 only, and the NUMA node target was toggled between local and remote to measure the difference.

The following assumptions are made for the environment using a 2P Intel system with 8 real cores per socket, but with Hyper-Threading disabled for this particular example. Examining the output of numactl –hardware, we would focus on three specific lines of output for this binding example.

available: 2 nodes (0-1)

node 0 cpus: 0,2,4,6,8,10,12,14

node 1 cpus: 1,3,5,7,9,11,13,15

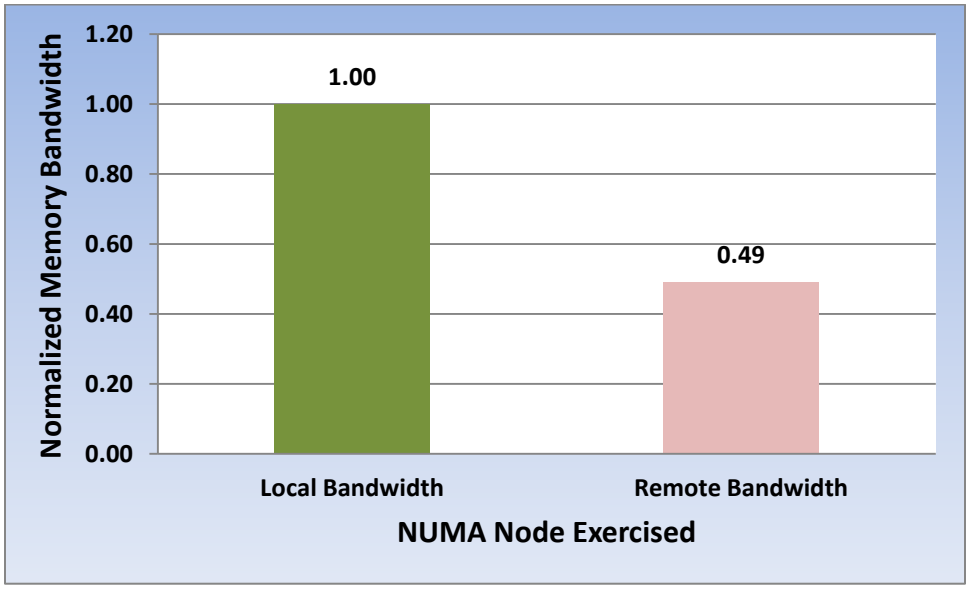
Figure 17 shows an example STREAM command for this example system to measure local NUMA node memory bandwidth. This example specifies memory allocation to NUMA node 0 using the –m 0 flag. This is considered **local** to cores 0,2,4,6,8,10,12,14 (assuming 8 threads for STREAM, one per physical core on socket 0, bound to cores local with NUMA node 0 with the --cpunodebind option):

numactl –m 0 --cpunodebind=0 stream

An example STREAM command to measure remote NUMA node memory bandwidth is shown in Figure 17. It specifies memory allocation to NUMA node 1 using the –m 1 flag. This is considered **remote** to cores associated with NUMA node 0 (assuming 8 threads for STREAM, one per physical core on socket 0, bound with the --cpunodebind option):

numactl –m 1 --cpunodebind=0 stream

Figure 17. Local vs. remote NUMA node memory bandwidth (higher is better)



In Figure 18, the relationship between local and remote NUMA node access is examined from the perspective of memory latency. In this case, remote memory latency is over 50% higher (worse) than local memory latency.

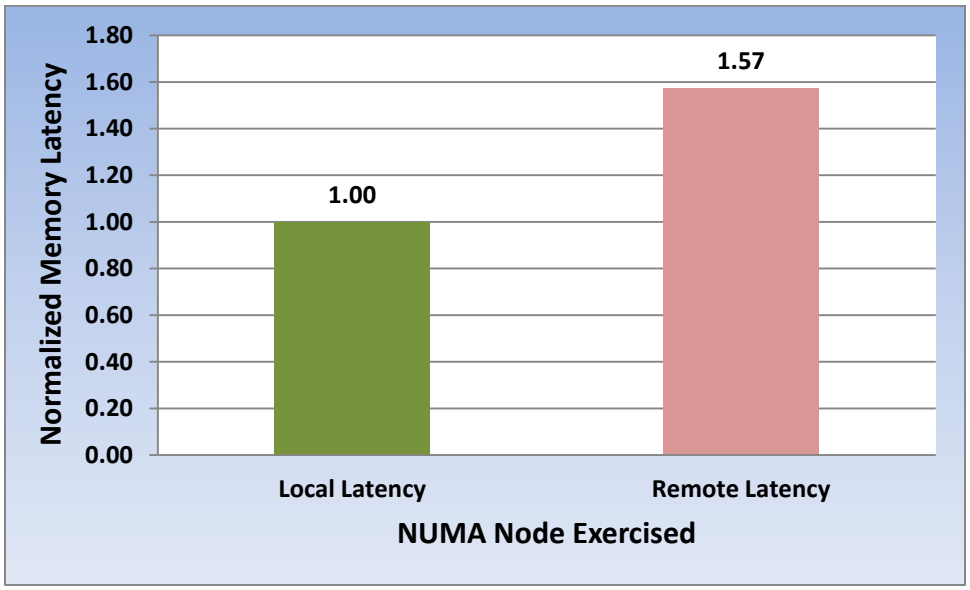
In the following example, the **lat_mem_rd** command is used to measure local NUMA node memory latency (NUMA node 0 is local to OS CPU 0) by specifying "–m 0". The -l flag could be used interchangeably with "-m 0" in this scenario.

numactl –m 0 –physcpubind=0 lat_mem_rd <size in MB> <stride size>

In the following example, lat_mem_rd command is used to measure remote NUMA node memory latency (NUMA node 1 is remote from OS CPU 0) by specifying "-m 1":

numactl –m 1 –physcpubind=0 lat_mem_rd <size in MB> <stride size>

Figure 18. Local vs. remote NUMA node memory latency (lower is better)



SPEC CPU2006 NUMA measurements

In Figure 19, we see various fully-multithreaded integer workloads run with explicit affinity using numactl. All workloads were part of SPECint_rate, and were initially executed explicitly bound to the correct logical processors and local NUMA nodes. The workloads were run again, and numactl was used to explicitly bind to same set of logical processors, but force memory allocation to the remote NUMA node for each thread. Examining the illustration, we can see that the impact of running a fully multithreaded workload with all remote memory access ranges from a negative impact of 3% with 400.perlbench to a high of 40% with the 429.mcf workload. Given the wide range of negative impacts, it can be concluded that being forced to run an application with all remote memory allocations will very likely result in a performance reduction. How much of a reduction depends on the workload and the sensitivity to increased memory latencies. These experiments were conducted on a 2P Intel platform, and care should be taken not to extrapolate these effects for 4P systems, which have different NUMA characteristics.

Figure 19. Local vs. remote NUMA binding – integer workloads

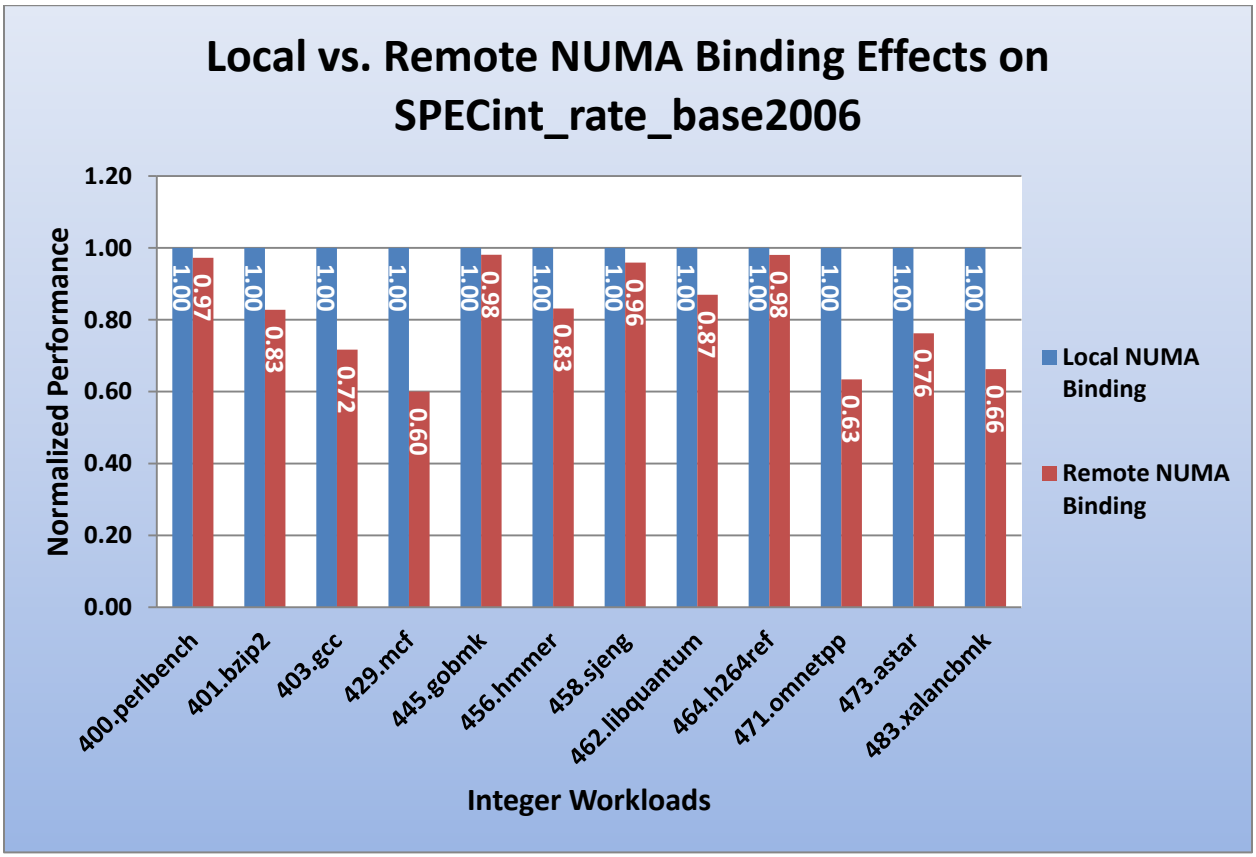
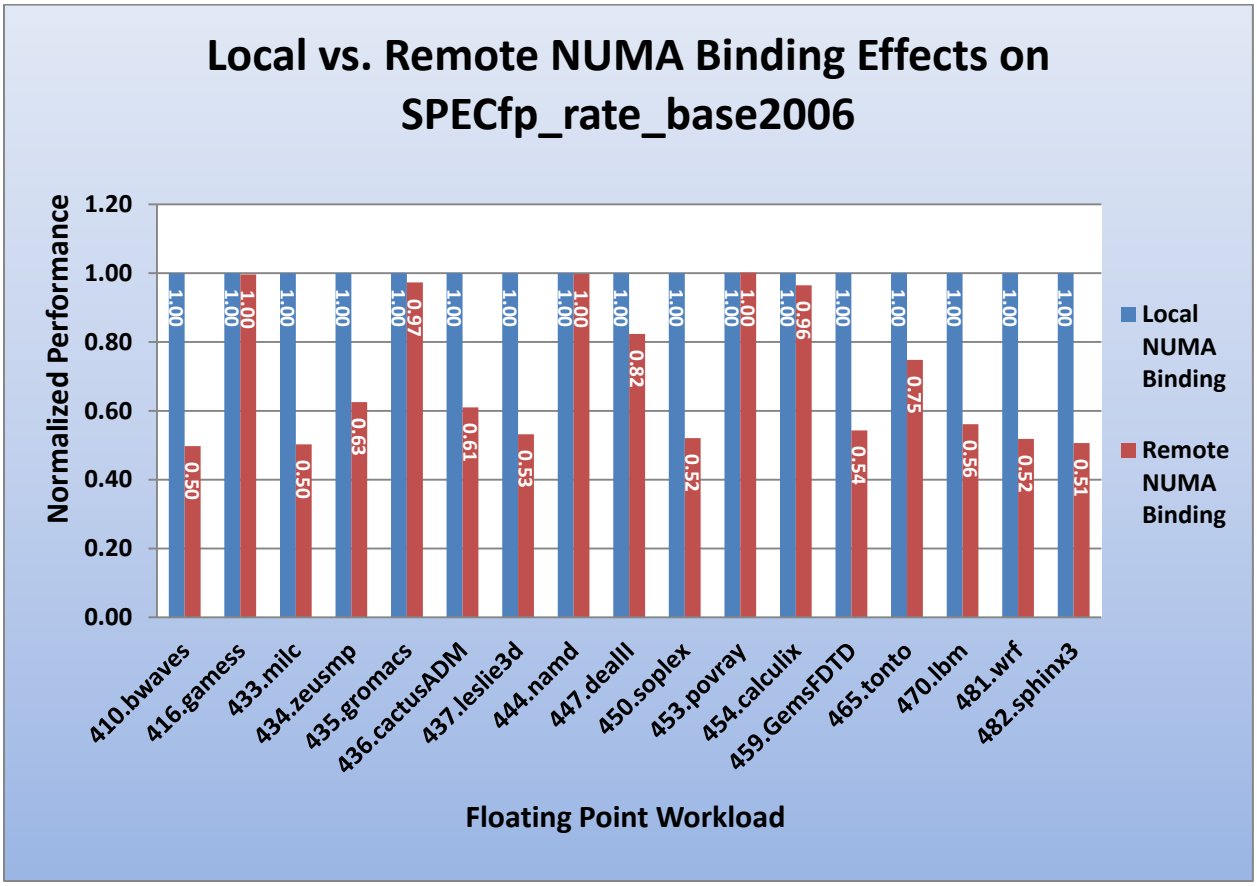


Figure 20 compares the effects of local and remote NUMA node affinity on floating point workloads contained in the SPECfp_rate benchmark. This benchmark's workloads are primarily scientific in nature, and tend to be more sensitive to memory bandwidth and memory latency. We can see a wide range of impacts in this comparison. Interestingly, three workloads experienced no performance loss when memory allocation was forced to the remote NUMA node. Other workloads were much more sensitive, and experienced up to 50% reduction in performance.

Based on the integer and floating point workloads explored, we can conclude that your mileage may vary in terms of potential performance loss when forcing memory to run remotely. Based on the limited subset of workloads comprising these two benchmarks, it appears that the effects are generally negative in nature and it should be inferred that local memory allocation is always preferred if available. There are conceivable situations where a new process or application must be started, but there is no free memory in the NUMA node local to the free logical processors intended to be utilized. The **numactl --hardware** command can be helpful in planning memory and core allocations for situations of this nature.

Figure 20. Local vs. remote NUMA binding – floating point workloads



NUMA I/O concepts

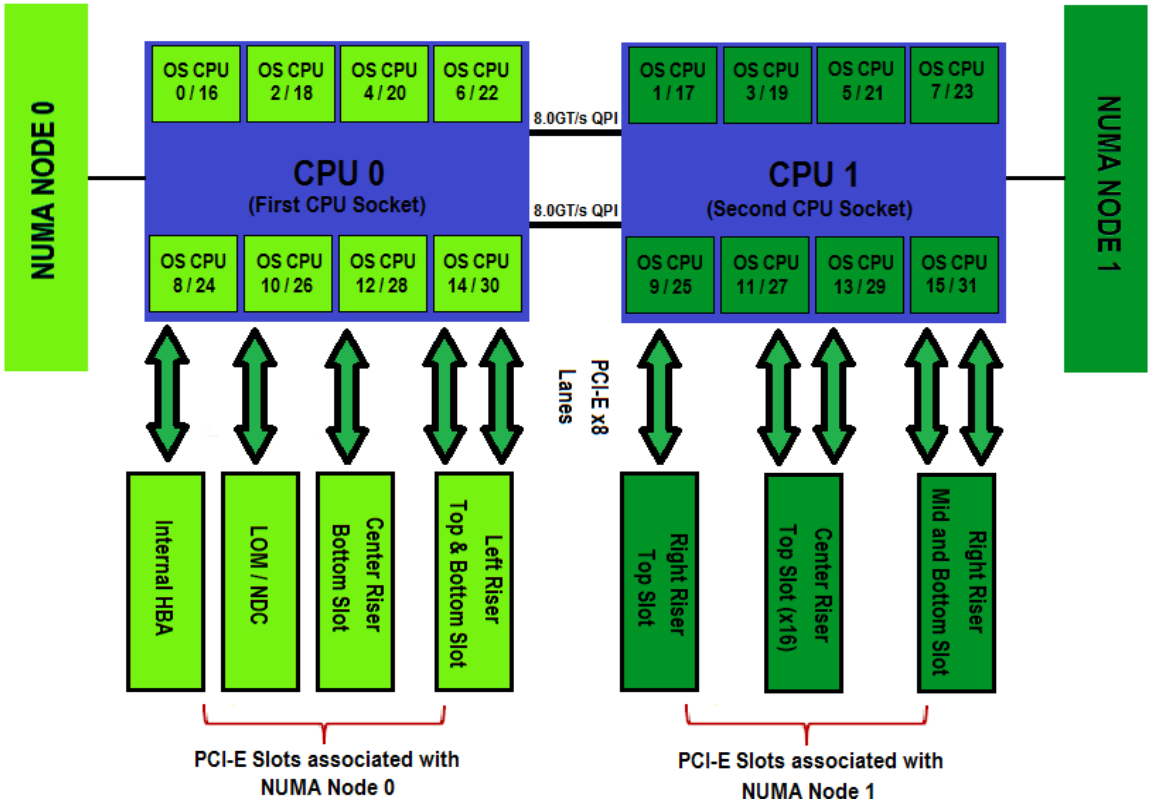
NUMA I/O is the idea of being able to localize memory access for I/O devices that are of varying distances away from processors and memory. The concept of NUMA I/O is relatively new to the x86 server architecture; Dell PowerEdge 12th generation servers supported NUMA I/O from the first day of launch for these platforms, which was not the case for all competitor server offerings.

The server BIOS provides information to the operating system to allow it to determine each PCIe device's NUMA node locality. The benefit of NUMA I/O most directly impacts heavy I/O or latency-sensitive computing environments. As this is a relatively new concept, several tools have been used to attempt to characterize NUMA I/O and establish some minimum best practices. However, this is an evolving field and considerable change can be expected in I/O driver NUMA awareness and additional tool features that may modify the best practices in the future.

Sample NUMA I/O topologies

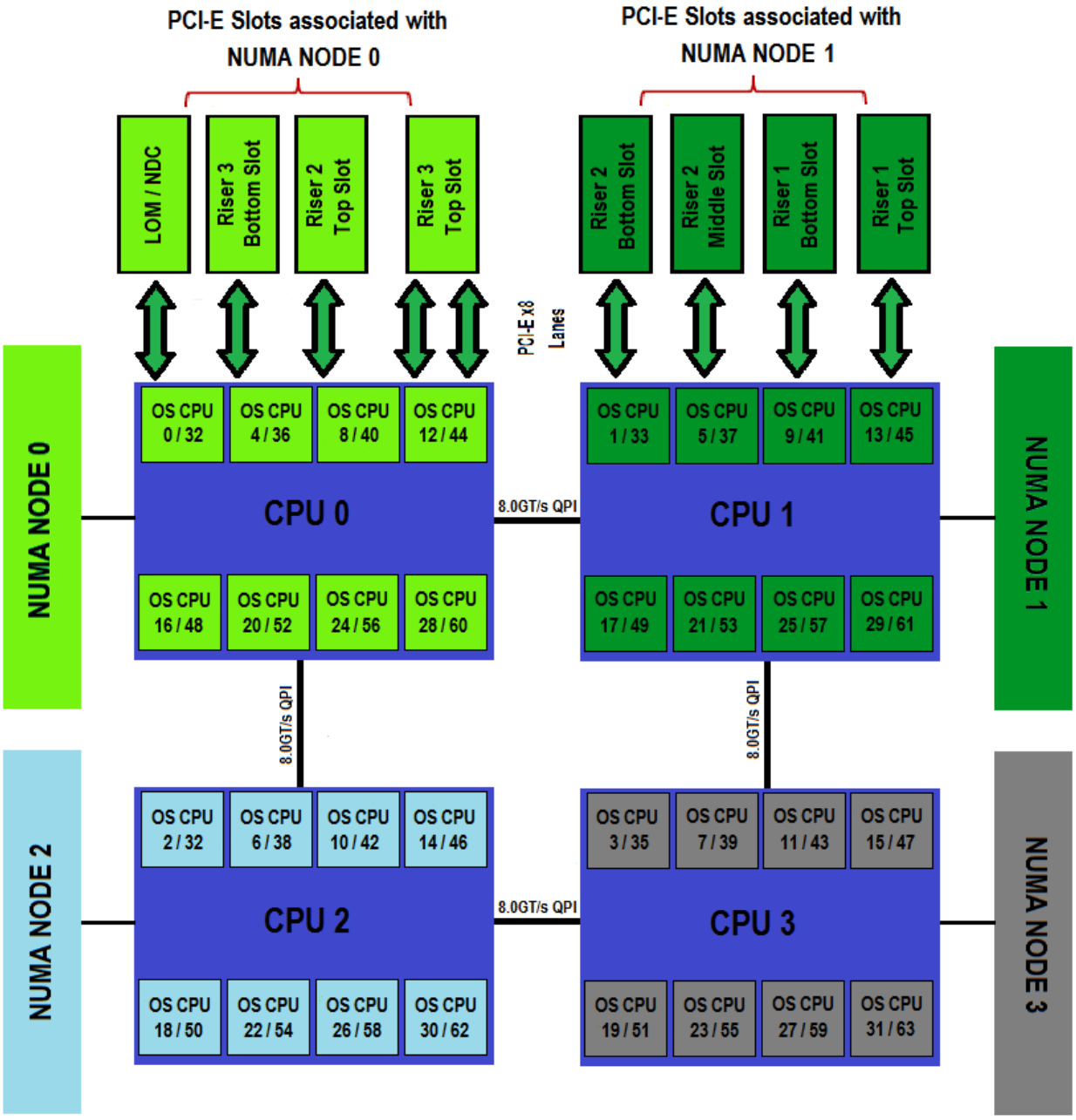
In Figure 21, we see a sample high-level NUMA I/O topology for a PowerEdge R720 with two E5-2600 8-core processors. The individual PCIe lanes are mapped to specific PCIe slots, and the relative NUMA node locality is described. All PCIe slots colored **light green** are part of NUMA node 0 in this example, and those in **dark green** are part of NUMA node 1.

Figure 21. 2P PowerEdge R720 NUMA I/O topology



In Figure 22, we see the NUMA I/O high-level topology for a PowerEdge R820 with four processors. We see the familiar CPU/core enumeration layout for a 4P E5-4600, but additionally we see particular PCIe lanes connected to CPU 0 and CPU 1; note that no PCIe lanes are immediately adjacent (local) to CPU 2 or 3.

Figure 22. 4P PowerEdge R820 NUMA I/O topology



Determination of PCIe NUMA locality

The first step to determine what NUMA node is local to a particular PCIe device is to examine the output of the **lspci -v** command. You can either redirect the output to a file or pipe it to the command "less", and search for the controller you are trying to identify. Note the PCI device number associated with the controller of interest. Figure 23 shows a snippet of the **lspci –v** output where two controllers of interest are located.

In Figure 23, we see two adapters snipped from a much larger list of PCI devices. The two devices are a PowerEdge RAID Controller H810 and a Mellanox high-speed Ethernet adapter. We can glean important information for these two adapters. The PCI device number for each adapter is listed at the upper left portion of the output for each PCI device entry. The PCI device number for the H810 in this specific example is **41:00.0**, and the Mellanox is **42:00.0**. The name of the device driver can be determined for each adapter from the same output at the bottom of each PCI device entry. The kernel driver for the H810 is **megaraid_sas** and the Mellanox Ethernet adapter uses **mlx4_core** and **mlx4_en**.

Figure 23. Example of lspci –v output

```
41:00.0 RAID bus controller: LSI Logic / Symbios Logic MegaRAID SAS 2208 [Thunderbolt] (rev 01)
        Subsystem: Dell PERC H810 Adapter
        Flags: bus master, fast devsel, latency 0, IRQ 66
        I/O ports at 7c00 [size=256]
        Memory at d4ffc000 (64-bit, non-prefetchable) [size=16K]
        Memory at d4f80000 (64-bit, non-prefetchable) [size=256K]
        Expansion ROM at d4000000 [disabled] [size=128K]
        Capabilities: [50] Power Management version 3
        Capabilities: [68] Express Endpoint, MSI 00
        Capabilities: [d0] Vital Product Data
        Capabilities: [a8] MSI: Enable- Count=1/1 Maskable- 64bit+
        Capabilities: [c0] MSI-X: Enable+ Count=16 Masked-
        Capabilities: [100] Advanced Error Reporting
        Capabilities: [1e0] #19
        Capabilities: [1c0] Power Budgeting <?>
        Capabilities: [190] #16
        Capabilities: [148] Alternative Routing-ID Interpretation (ARI)
        Kernel driver in use: megaraid_sas
        Kernel modules: megaraid_sas

42:00.0 Ethernet controller: Mellanox Technologies MT27500 Family [ConnectX-3]
        Subsystem: Mellanox Technologies Device 0029
        Flags: bus master, fast devsel, latency 0, IRQ 72
        Memory at d5f00000 (64-bit, non-prefetchable) [size=1M]
        Memory at d0000000 (64-bit, prefetchable) [size=8M]
        Expansion ROM at d5000000 [disabled] [size=1M]
        Capabilities: [40] Power Management version 3
        Capabilities: [48] Vital Product Data
        Capabilities: [9c] MSI-X: Enable+ Count=128 Masked-
        Capabilities: [60] Express Endpoint, MSI 00
        Capabilities: [100] Alternative Routing-ID Interpretation (ARI)
        Capabilities: [148] Device Serial Number 00-02-c9-03-00-45-22-50
        Capabilities: [154] Advanced Error Reporting
        Capabilities: [18c] #19
        Kernel driver in use: mlx4_core
        Kernel modules: mlx4_en, mlx4_core
```

Another method of locating the PCIe device number to use is searching by slot number. This can be accomplished using the following:

dmidecode -t slot: Look for slots "In Use", and obtain the **Bus Address**.

lspci -s <Bus Address>: This command gives you the PCIe device number for the selected slot obtained with the previous command.

Next, the PCI device identified must be located in **/sys/devices/pci***, and the **numa_node** file must be examined to determine the local NUMA node for this device. Figure 24 shows an example where the PERC is identified and the local NUMA node is noted. As we can see from the output, this PCIe device is associated with NUMA node 1 (local to the second processor socket in a 2P Intel platform).

Figure 24. Example of deriving PCIe device NUMA locality

```
[root@system /]# cd /sys/devices
[root@system devices]# ls
cpu  LNXSYSTM:00  pci0000:00  pci0000:3f  pci0000:40  pci0000:7f  platform  pnp0  software  system  tracepoint  virtual
[root@system devices]# cd pci0000\:40
[root@system pci0000:40]# ls
0000:40:01.0  0000:40:02.0  0000:40:03.0  0000:40:03.2  0000:40:05.0  0000:40:05.2  firmware_node  pci_bus  power  uevent
[root@system pci0000:40]# cd 0000\:40\:01.0/
[root@system 0000:40:01.0]# ls
0000:40:01.0:pcieffffffc1  broken_parity_status  device  firmware_node  local_cpulist  msi_bus    power   resource         subsystem_vendor
0000:41:00.0               class                driver  irq            local_cpus     numa_node  remove  subsystem        uevent
acpi_index                 config               enable  label          modalias       pci_bus    rescan  subsystem_device  vendor
[root@system 0000:40:01.0]# cat numa_node
1
```

It is also important to be aware of where the device driver is handling interrupts for a given adapter. In Figure 25, we see sample output from the command **"cat /proc/interrupts"**. The leftmost column is the IRQ number, the middle columns are interrupt counts from each OS CPU, and the rightmost column shows the kernel module generating the interrupts for a particular row. As a driver generates interrupts, the corresponding interrupt count will increase for a given CPU.

Figure 25. /proc/interrupts example

```
[root@system ~]# cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7       CPU8       CPU9       CPU10      CPU11      CPU12      CPU13      CPU14      CPU15
  0:        163          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   IO-APIC-edge      timer
  3:          1          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   IO-APIC-edge
  4:          1          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   IO-APIC-edge
  8:          1          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   IO-APIC-edge      rtc0
  9:          2          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   IO-APIC-fasteoi   acpi
 22:        937          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   IO-APIC-fasteoi   ehci_hcd:usb2
 23:        113          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   IO-APIC-fasteoi   ehci_hcd:usb1
 88:        890          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   HPET_MSI-edge     hpet2
 89:          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   HPET_MSI-edge     hpet3
 90:          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   HPET_MSI-edge     hpet4
 91:          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   HPET_MSI-edge     hpet5
 92:          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   HPET_MSI-edge     hpet6
105:      25843          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   PCI-MSI-edge      megasas
106:    3247693          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   PCI-MSI-edge      megasas
107:          1          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   PCI-MSI-edge      em4
108:     143797          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   PCI-MSI-edge      em4-TxRx-0
115:          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   PCI-MSI-edge      eth-mlx4-0
116:          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   PCI-MSI-edge      eth-mlx4-1
117:          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   PCI-MSI-edge      eth-mlx4-2
118:    4272085          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   PCI-MSI-edge      mlx4_core(async)
NMI:          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   Non-maskable interrupts
LOC:    3059970    2305953     296147     366634     297567     325041     281173     283758     910420     327059     300810     366415     361233     327292    3090658     287327   Local timer interrupts
SPU:          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   Spurious interrupts
PMI:          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   Performance monitoring interrupts
PND:          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   Performance pending work
RES:      38001      35434      41061      34752      48817      33675      33627      33588      43685      34876      35002      42476      48179      34028    1822488      33688   Rescheduling interrupts
CAL:       4018       3491       2152        569        964      25601       1921        218       4036      51452        202       2658       2461      38815        208       3267   Function call interrupts
TLB:        491        353       2284        189        151        125        113         97        173        217        816        759       4984        275        151        134   TLB shootdowns
TRM:          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   Thermal event interrupts
THR:          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   Threshold APIC interrupts
MCE:          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0          0   Machine check exceptions
MCP:        892        892        892        892        892        892        892        892        892        892        892        892        892        892        892        892   Machine check polls
ERR:         15
MIS:          0
```

For a system with many logical processors and PCI devices, it may be preferable to run the command "**cat /proc/interrupts | grep <driver name>**" to pull only IRQ lines of interest.

irqbalance is a Linux daemon (running by default in many cases) that distributes interrupts for devices across cores to prevent particular logical processors from being overrun with work. However, it may be advisable under extremely heavy I/O conditions to manually affinitize a particular device's interrupts to a specific core. **Note that more recent drivers often use the MSI-X interrupt model by default, which provides a larger number of interrupt vectors. Each vector is capable of being individually bound. These examples are from an adapter set to use the standard IRQ model**. To manually affinitize an IRQ to a specific core, one must stop the **irqbalance** daemon from running.

service irqbalance stop stops a running irqbalance daemon

chkconfig irqbalance off prevents the irqbalance daemon from starting at next boot

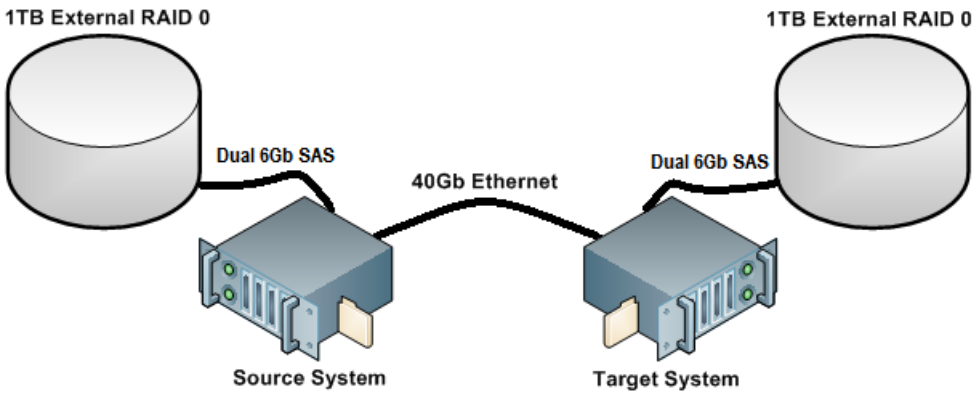
For additional information regarding the process for manually setting interrupt affinity, please look here and here.

Multiple I/O devices under heavy load – Netcat

One method to characterize I/O characteristics in a NUMA environment for this white paper was to configure two Dell PowerEdge R720 systems, each with an external RAID adapter connected to a separate 1TB storage array. Both systems also contained a Mellanox® 40Gb Ethernet adapter, which was directly connected from one system to the other. The purpose of this exercise was to evaluate the impact of IRQ binding and I/O worker thread affinity on throughput characteristics.

A series of files ranging in size from 512MB to 500GB were timed as they were transferred from one server to another across the 40Gb Ethernet link and written to external storage on the other end. Experiments were conducted to evaluate the efficacy of irqbalance in determining the appropriate core as well as the impact of binding worker threads on both systems in various configurations to determine the impact on file transmission. The Linux **netcat** tool was used to transmit the files across the Ethernet from the sender system and write them on the receiver system. Figure 26 describes the hardware configuration for this test.

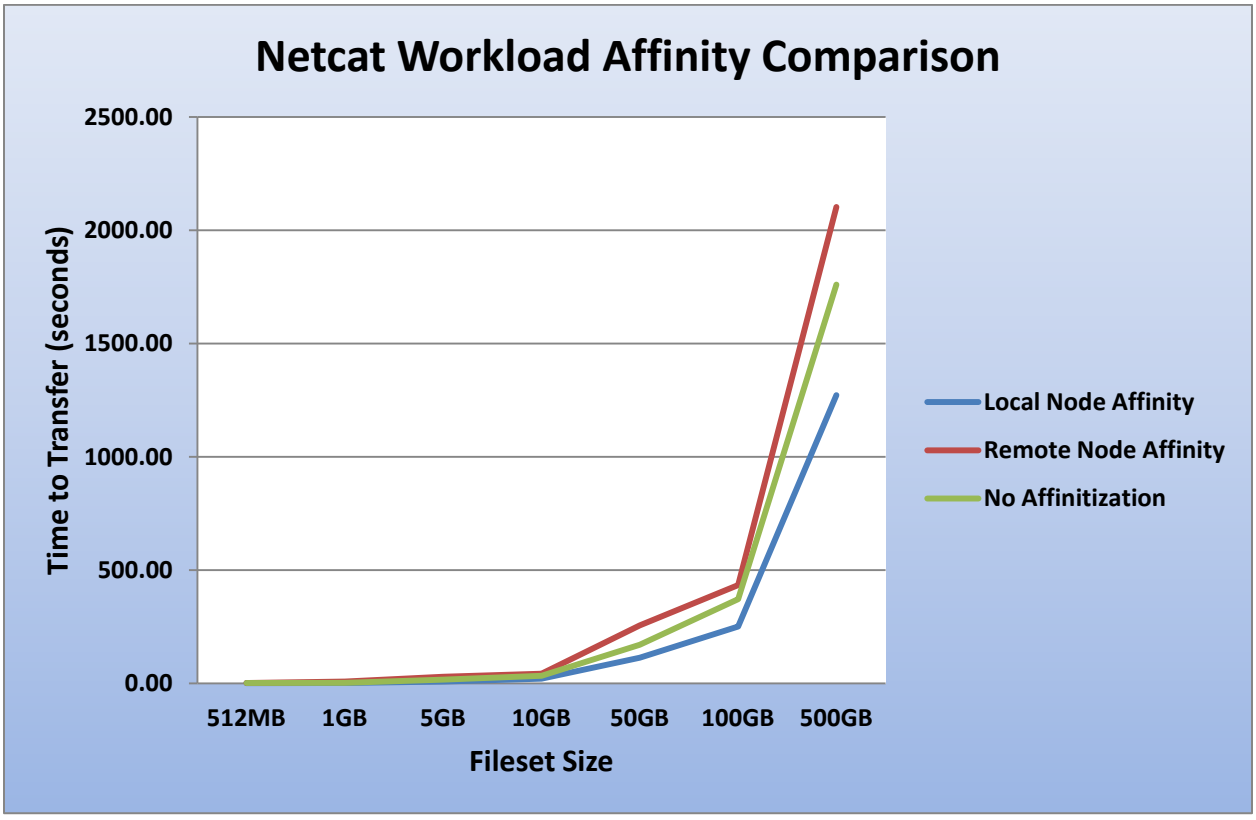
Figure 26. Netcat workload diagram



During the course of testing, the version of **irqbalance** that was distributed with RHEL 6 was updated to the newest version available, which provided equivalent performance to manual IRQ binding of the Mellanox and PERC interrupts. Although in some circumstances it may still be better to manually affinitize IRQs to particular cores, this test suggested that in that the new irqbalancer was sufficient, even under heavy I/O conditions for this test. However, manual interrupt binding should always be considered when maximum performance or lowest latency is required.

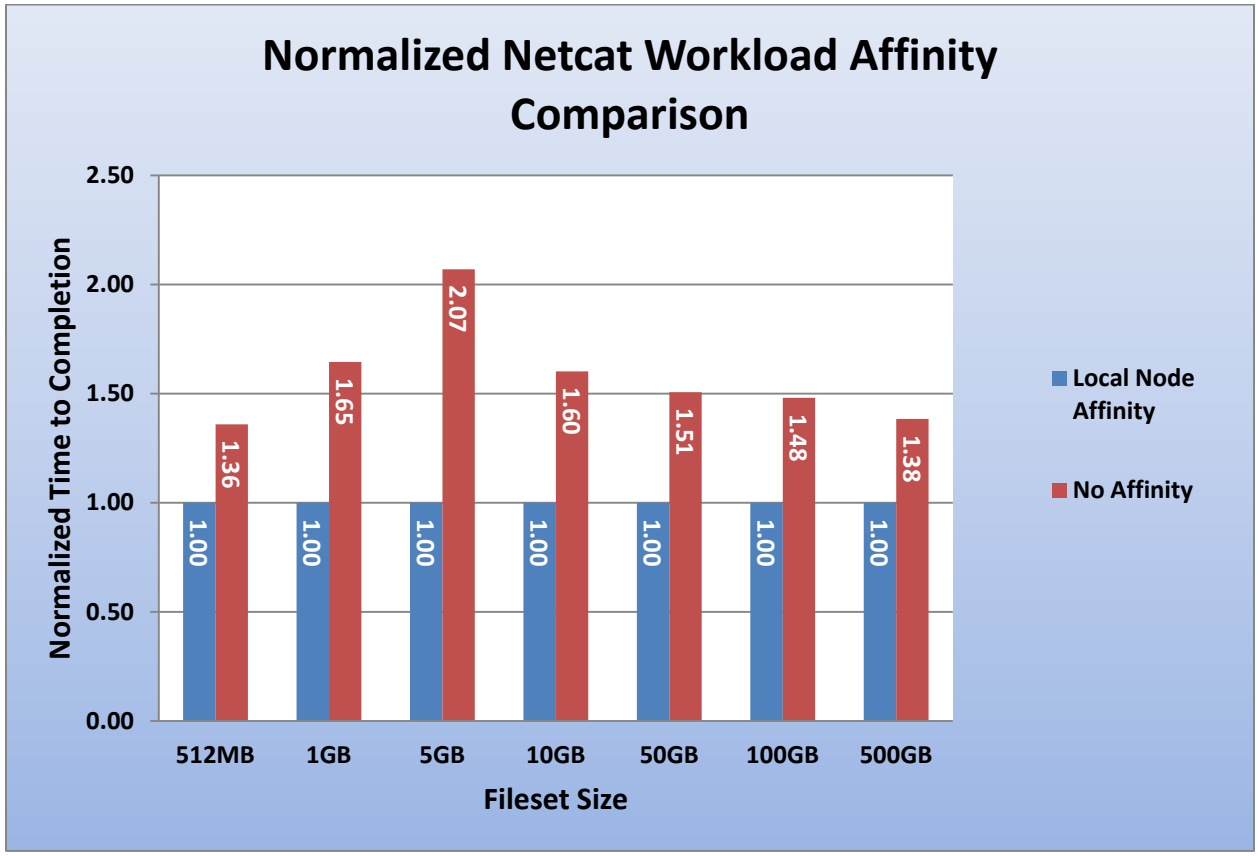
Several experiments were conducted with **netcat** to time the transmission various files as they were read from the sender system, sent across the Ethernet link to the receiver system, and written out to the remote disk array. Both systems were configured with 40Gb Ethernet and external RAID adapters in PCIe slots local to processor socket 1. IRQs for both devices were manually bound to cores on socket 1, and the netcat worker on sender and receiver were switched between binding to memory and cores local to socket 0, socket 1, and run without affinity. Each test was run for three iterations, and the median times for each file size were reported in Figure 27. As you can see, binding to the local NUMA node using cores on the same socket that the IRQ was bound provided **consistently shorter transmission times** than either not affinitizing the worker processes or binding them to remote memory and cores.

Figure 27. Netcat workload affinity comparison



In Figure 28, we see a similar experiment where local affinity of worker threads was compared to no specified affinity. In this case, not affinitizing sender and receiver threads resulted in consistently slower transmission times, up to twice as long for some file sizes.

Figure 28. Normalized netcat comparison – local vs. no affinity



Maximum IOPS scenarios using Vdbench

Vdbench is a Java-based I/O measurement tool that is growing in popularity in the Linux world due to its ease of use and repeatability of results. This tool was used to measure the impact of affinitizing the worker JVMs and memory to the same or different NUMA nodes relative to the fixed IRQ assignment for the RAID adapter. For this experiment, a system was configured with 8 x SAS SSD drives in a RAID 0 array. This storage configuration offered the highest possible IOPS to be driven to simulate an I/O-bound environment. Most real world storage configurations and workload patterns will not approach this configuration, but for the purposes of this test it was an interesting example of the effects of I/O worker affinity and memory allocation relative to the IRQ.

Figure 29 illustrates the performance impact of affinitizing the workers and memory allocation to the local versus remote NUMA nodes. The configuration that bound worker JVMs and memory allocation to the same NUMA node as the IRQ achieved significantly higher IOPS, as opposed to being bound to a remote NUMA node, which was only able to achieve 63% of the optimal IOPS.

Figure 29. IRQ/workload location comparison - Vdbench

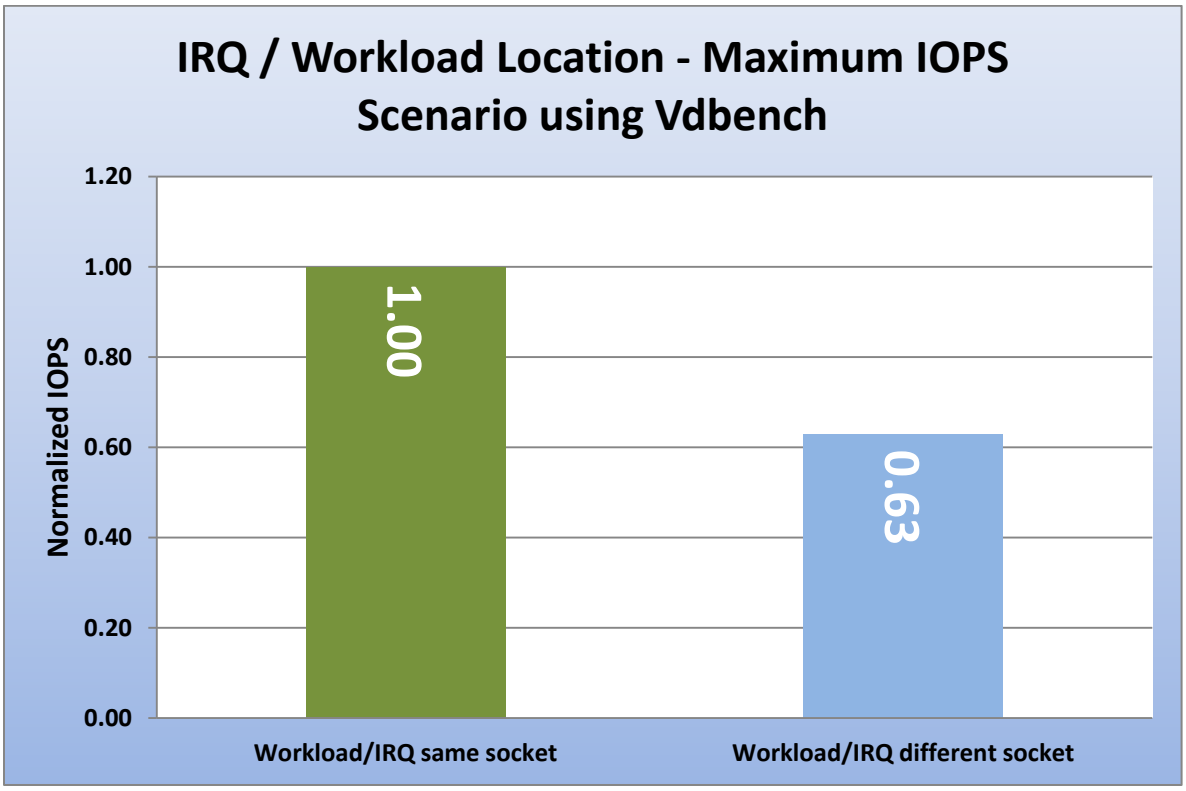
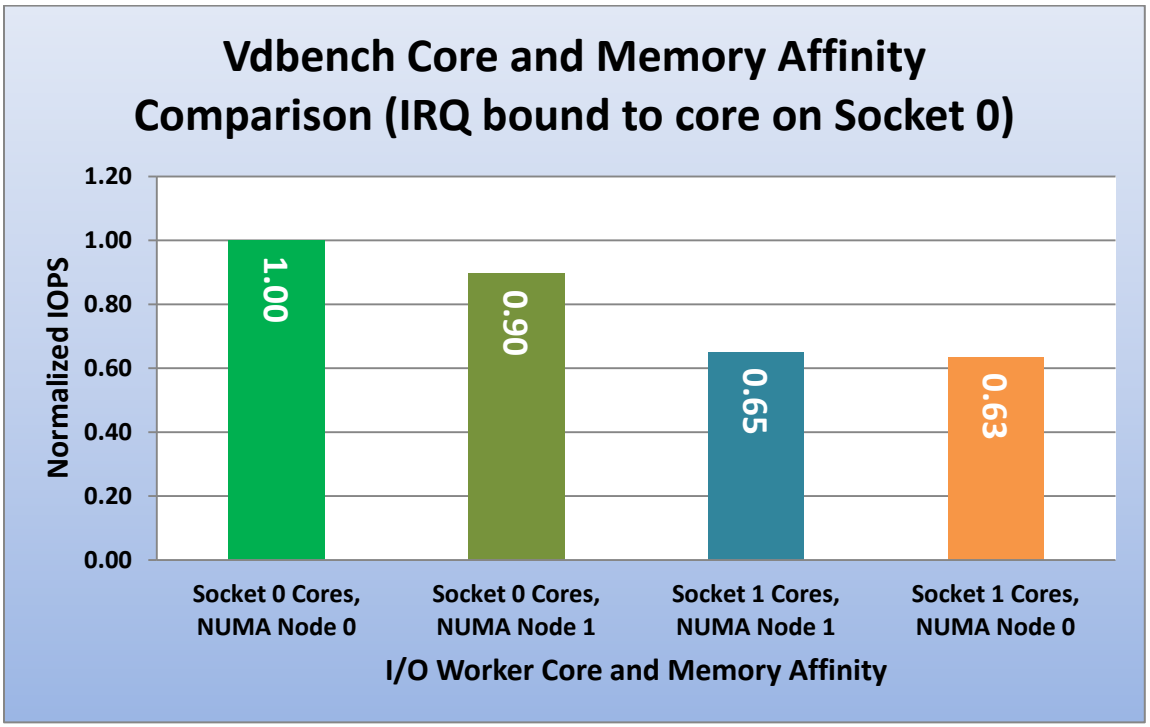


Figure 30 shows the effects of binding the Vdbench I/O worker JVMs to cores and NUMA nodes independent of one another. Manually affinitizing the worker JVMs to the same socket and NUMA node that the IRQ is bound provided the highest overall IOPS. Keeping the JVMs bound to the same socket but forcing allocation to the remote NUMA node achieved only 90% of the total IOPS of the ideal configuration. Worker JVMs were then bound to the opposite socket and NUMA node to where the RAID adapter was bound, which only yielded 65% of the IOPS that the optimum binding/affinity combination. The JVMs were then forced to allocate memory to the opposite NUMA node, and performance in this case dropped to only 63% of the optimum affinitization.

Figure 30. Core and memory affinity effects - Vdbench



The new frontier of NUMA I/O and numactl

Some interesting new NUMA I/O features have recently been added to the **numactl** command in version 2.0.8. For common numactl flags such as --membind or --cpubind, the user can now supply one of several options to bind a program to the NUMA node local to the underlying I/O adapter's PCIe address that is referenced.

- **Network Interface** name (eth0, em0, etc.) using *netdev:<device name>*
- **Local IP Address** using *ip:<ip address>*
- **File Location** using *file:</path/file>*
- **Block Device** using *block:<block device>*
- **PCI Device Number** using *pci:<pci device number>*

Conclusion

Although Linux server-oriented operating systems have become increasingly NUMA-aware, complete NUMA awareness has not penetrated every level of the application, kernel, driver, and tool stack. Modern Linux distributions provide acceptable performance for many general computing environments without forcing the user to manually affinitize every element. However, for heavily loaded systems or for systems where maximum throughput or minimum latency is a focus, the process of manual affinitization can be beneficial.

NUMA I/O will be of increasing importance as I/O devices evolve to higher speed interconnects with storage and networking. For maximum I/O throughput scenarios, you should be aware of where I/O worker threads are running in relation to the IRQ, and attempt to force memory allocation to local NUMA nodes for maximum throughput or latency characteristics.

References

RHEL 6 Performance Tuning Guide: http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/index.html

Linux Memory Management: http://www.tldp.org/LDP/tlk/mm/memory.html

Memory NUMA Support: http://lwn.net/Articles/254445/

Automatic Page Migration for Linux: http://lca2007.linux.org.au/talk/197.html

Red Hat Introduction to Control Groups: https://access.redhat.com/knowledge/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html

SUSE Cpuset Tutorial: https://www.suse.com/documentation/slerte_11/slerte_tutorial/data/slerte_tutorial.html

Netcat: http://netcat.sourceforge.net/

Message Signaled Interrupts (MSI-X): http://en.wikipedia.org/wiki/Message_Signaled_Interrupts

Interrupts and IRQ Tuning: https://access.redhat.com/knowledge/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-cpu-irq.html

SMP IRQ Affinity: https://cs.uwaterloo.ca/~brecht/servers/apic/SMP-affinity.txt

Vdbench: http://sourceforge.net/projects/vdbench/

Dell 12G BIOS Tuning white paper: http://en.community.dell.com/techcenter/extras/m/white_papers/20248740.aspx

Dell 12G Memory Performance Guidelines white paper: http://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/12g-memory-performance-guide.pdf