Hewlett Packard Enterprise

Technical white paper

Red Hat Enterprise Linux NUMA support for HPE ProLiant servers

Red Hat Enterprise Linux 5.x, 6.x, and 7.x

Contents

Abstract

This white paper discusses Linux® support for HPE ProLiant servers with non-uniform memory access (NUMA) architecture design as delivered in the Red Hat® Enterprise Linux (RHEL) and other Linux distributions. To optimize performance, Linux provides automated management of memory, processor, and I/O resources on most NUMA systems. However, as servers scale up in both processor speed and processor count, your ability to obtain optimum performance for large, long-lived enterprise and technical computing applications requires an understanding of how Linux manages these resources on a NUMA system.

This white paper is intended for experienced HPE and customer personnel who design application systems, analyze and optimize the performance of those applications on HPE ProLiant servers with NUMA running Linux. The information in this white paper applies to applications that are NUMA-aware and to those that are not. Readers are expected to have an in-depth understanding of multi-processor computer systems and operating systems, including basic virtual memory concepts.

NUMA

The Red Hat Enterprise Linux kernel gained support for cache-coherent NUMA systems in the Red Hat Enterprise Linux kernel 2.6 release series. The kernel's support for NUMA system has continued to evolve over the lifespan of the 2.6 and 3.x kernels, and now includes a significant subset of the NUMA features expected in an enterprise-class operating system. Enterprise distributions such as Red Hat Enterprise Linux 6 and 7 acquire these features in major releases and updates. As a result, for many NUMA servers and workloads, the kernel achieves near-optimum performance so that NUMA-specific tuning is seldom required.

For situations where the kernel's default NUMA behavior is inappropriate or suboptimal, Linux provides a rich collection of commands and application programming interfaces to give an administrator or programmer some control over system behavior. Linux also provides information showing the memory usage and page placement of tasks resulting from the kernel's memory allocation and placement decisions.

This document explains default Linux NUMA behavior and discusses the tools available to examine and affect this behavior. This section discusses the benefits of using NUMA servers, as well as the implications of their use for both kernel and application software.

Hewlett Packard Enterprise incorporates the NUMA architecture in its servers primarily because of the architecture's ability to provide scalable memory and I/O bandwidth for today's high-core-count, high-performance processors, within the constraints of physical and economic feasibility. In fact, with memory controllers moving "on-chip" in most contemporary commodity x86-64 processors, most multi-socket servers are now inherently NUMA systems.

Introduction to NUMA topology

A NUMA server can be viewed as a collection of SMP/SMT building blocks, each containing a subset of the server's processors, memory, and I/O connections. These building blocks are often referred to as nodes or cells—in Red Hat Enterprise Linux kernel terminology, they are referred to as nodes. This white paper also refers to them as such. Figure 1 shows an abstract representation of a simple two-node NUMA server. The nodes are connected by a system interconnect link.

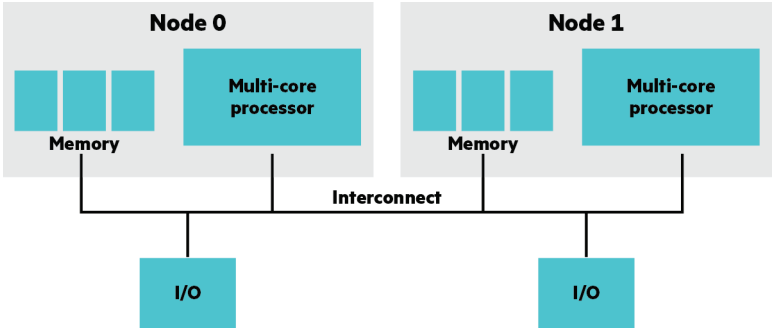


Figure 1. Simple two-node NUMA server

Ideally, the intra-node core-to-memory and I/O-to-memory bandwidths are sufficient to handle:

(1) The requirements of the cores and the I/O subsystems contained within each node and

(2) Incoming accesses from remote nodes.

If the intra-node bandwidth is insufficient for the node's core and I/O requirements, the server suffers from the same bottlenecks within a node that the NUMA implementation is intended to address at the system level.

Within cost and power constraints, the inter-node interconnect should have the lowest latency and highest bandwidth possible to handle off-node references made by the nodes. Although the interconnect bandwidth may approach or exceed the intra-node bandwidth, it is often incapable of supporting the maximum rate at which other nodes in the server can produce off-node references. (An off-node reference is a non-local reference—that is, a reference to memory existing on a node remote to the reference-issuing node.)

Interconnect can limit performance

Typically, the remote memory access latencies quoted in specification sheets are measured for uncontended accesses to a remote node. These latencies are often specified as ratios of remote memory/access memory latency to local memory latency, and they often represent best-case scenarios.

As the load on an interconnect increases, contention for access to the interconnect can increase latency beyond these quoted values. For example, if multiple applications simultaneously generate streams of remote accesses on the same interconnect, request arbitration and the queuing overhead for managing each stream increases, in turn increasing the average time required to perform a remote access. When the average access latency in a stream of accesses generated by an application increases, the result is a decrease in the memory bandwidth available to the application, which is likely to impede performance.

So, independent of the quoted ratio of local to remote latencies, the primary goal of the NUMA-aware software is to reduce the consumption of inter-node bandwidth by keeping the majority of memory accesses local to the requesting CPU or I/O bus. Local accesses incur lower local latencies and benefit from greater node-local bandwidth. The remaining off-node accesses experience less interconnect contention and incur latencies closer to the quoted best-case values.

Node locality is essential for performance

With modern high-clock-rate processors, performance greatly depends on caching behavior. With a high-cache hit rate, the processor performance nearly reaches its theoretical optimum; with a low-cache hit rate, performance suffers. In turn, cache efficiency depends on the locality of memory references. That is, processor caches improve performance when the program tends to access multiple words in a cache line soon after the cache line is first fetched from memory. If this is not the case, the processor unnecessarily consumes memory bandwidth by having to fetch the additional words of the cache line from memory.

NUMA servers depend similarly on locality to achieve the benefits of local memory bandwidth and latency. For example, if 100 percent of processor cache misses access-only local memory, the effective memory bandwidth seen by all processors equals the sum of the intra-node bandwidth of all the nodes. If processors on each server's nodes simultaneously attempt to access the memory on a single node, the effective memory bandwidth **cannot** exceed the intra-node bandwidth of that one node.

So, how do you maximize locality for NUMA servers? The next section provides answers.

Red Hat Enterprise Linux kernel default NUMA behaviors

Ideally, system software developers modify the kernel, libraries, compiler toolchains, and other components to achieve optimal performance by default in a wide array of circumstances. Primarily, this means providing reasonable (that is, non-pathological) default behaviors for the following:

- Selection of applications' execution CPUs and memory locality relative to those CPUs
- Design of kernel algorithms and data structures, as well as data placement
- Selection of kernel or application I/O paths

The Red Hat Enterprise Linux kernel's NUMA features evolved throughout the Red Hat Enterprise Linux 2.6 release series to implement reasonable defaults for these areas. With the 2.6.16 release, improved NUMA features began to provide significant performance benefits for many workloads by default. Application performance measured on Red Hat Enterprise Linux servers configured for NUMA often surpassed the performance measured on the same server configured to interleave memory pages across all its nodes in hardware.[1] Since then, as development has progressed to the current Red Hat Enterprise Linux 3.x releases, the default Red Hat Enterprise Linux NUMA behavior continues to improve and keeps pace with the newest NUMA server topologies.

However, since memory access patterns and the effects on application performance can be workload dependent, Red Hat Enterprise Linux allows you to tune its behavior for different workloads. For example, when the free memory on a given node is exhausted, should the next allocation request on that node attempt to reclaim pages on it to preserve locality—a potentially costly process? Or should the system allocate a page on a remote node for the application? A suitable answer depends on the behavior and lifetime of the application. To select between these behaviors, Red Hat Enterprise Linux provides a tunable parameter discussed in the "Using the `vm.zone_reclaim_mode`" section of this paper.

System tunable parameters such as `vm.zone_reclaim_mode` have global scope, so they affect all processes running on the system. On some servers, you might want to run multiple applications that have unique tuning requirements. Some applications might consist of multiple subsystems that in turn have their own unique tuning requirements. To address these more finely-grained tuning needs, Linux provides mechanisms that allow an administrator, end user, or programmer to customize some aspects of the kernel's NUMA behavior for individual tasks. An example is the memory allocation policy, which can be configured uniquely for different tasks and even for different memory regions. (For more information, refer to the "Red Hat Enterprise Linux memory policy principles" section of this document.)

Linux also provides mechanisms that can constrain the set of hardware resources an application can use. When combined with knowledge of a server's NUMA topology, these mechanisms can be used to run applications on the hardware where they will perform well and where detrimental performance interactions can be avoided. (For information about one of the ways to do this, refer to "Using cpusets [privileged users only] to Group Tasks and Constrain CPU/Memory Usage" section of this document.) These mechanisms fall into two categories:

- Inheritable task attributes (across `fork()/exec()`) or attributes that can be applied to a running task. You can use these attributes to optimize the NUMA behavior of unmodified binary applications using command line tools such as the `numactl` command (see the `numactl(8)` manpage).
- Application programming interfaces (APIs) (see the `set_mempolicy(2)` and `mbind(2)` manpages) for use by NUMA-aware applications and language run-time environments to specify the desired behavior, perhaps based on the available resources. These APIs require source modifications, recompilation, and/or relinking.

HPE servers with NUMA architecture

This section describes NUMA topology that can be found on HPE ProLiant servers and examines selected HPE ProLiant servers that fall in each kind of NUMA architecture. In addition, this section provides the context for understanding Linux and NUMA features, as well as introduces some commands and concepts explained in more detail in subsequent sections of this document.

Prevalence of the NUMA architecture

High-end enterprise and technical computing servers have long exhibited NUMA characteristics. High-end HPE PA-RISC/Intel® Itanium® and HPE Mission Critical Solutions use NUMA architecture to increase their memory scalability to support large numbers of high-performance processors. With the introduction of AMD processors containing HyperTransport (HT) links, commodity x86-64 two- and four-socket NUMA servers became commonplace. With the more recent availability of the QuickPath Interconnect (QPI), x86-64 servers built with Intel® processors now also exhibit NUMA characteristics. HPE ProLiant multi-socket servers based on Intel QPI or AMD HT processors are inherently NUMA systems.

[1] Interleaving memory pages across nodes roughly approximates some of the uniform memory access latency characteristics of a traditional SMP system. However, this technique does not hide the system's NUMA characteristics and users are often surprised by this.

Interconnect technology used on HPE ProLiant servers

As mentioned in Interconnect can limit performance section, the interconnect technology used could impact the performance and NUMA topology on HPE ProLiant servers. There are three interconnect technologies one could find on HPE ProLiant servers:

- Intel UltraPath Interconnect (UPI)

  Intel UPI is the interconnect technology used in the Intel® Xeon® Scalable processor family. UPI is a coherent interconnect for scalable systems containing multiple processors in a single shared address space. Intel Xeon processors that support UPI, providing either two or three UPI links for connecting to other Intel Xeon processors, do so using high-speed, low-latency path to the other CPU sockets. Intel UPI uses a directory-based home snoop coherency protocol, which provides an operational speed of up to 10.4 GT/s. It improves power efficiency through a low-power state, provides improved data transfer efficiency over the link using a new packet format, and has improvements at the protocol layer such as no preallocation to remove scalability limits with Intel QPI.

  HPE ProLiant Gen10 Servers with Intel Xeon Scalable processors use UPI as its interconnect technology. Depending on the model, processors may support two or three UPI links. The number of UPI links decides the NUMA topology and inter-node performance.

  - Two UPI Links
    - Systems with processors that support two UPI links can only be ring topology in NUMA and may introduce extra latency when accessing remote memory. Intel Xeon Gold 5100 Processor series and below are processors of this kind.
  - Three UPI Links
    - Systems with processors that support three UPI links can be fully connected in NUMA. Intel Xeon Gold 6100 Processor series and Intel Xeon Platinum Edition are processors of this kind.

- Intel QuickPath Interconnect

  Intel QPI is a high-speed, packetized, and point-to-point interconnect used in Intel Xeon processors prior to the Intel Xeon Scalable family. Intel Xeon processors that support QPI, providing either two or three QPI links for connecting to other Intel Xeon processors and one or more IO hubs or more routing hubs in the network on the system board.

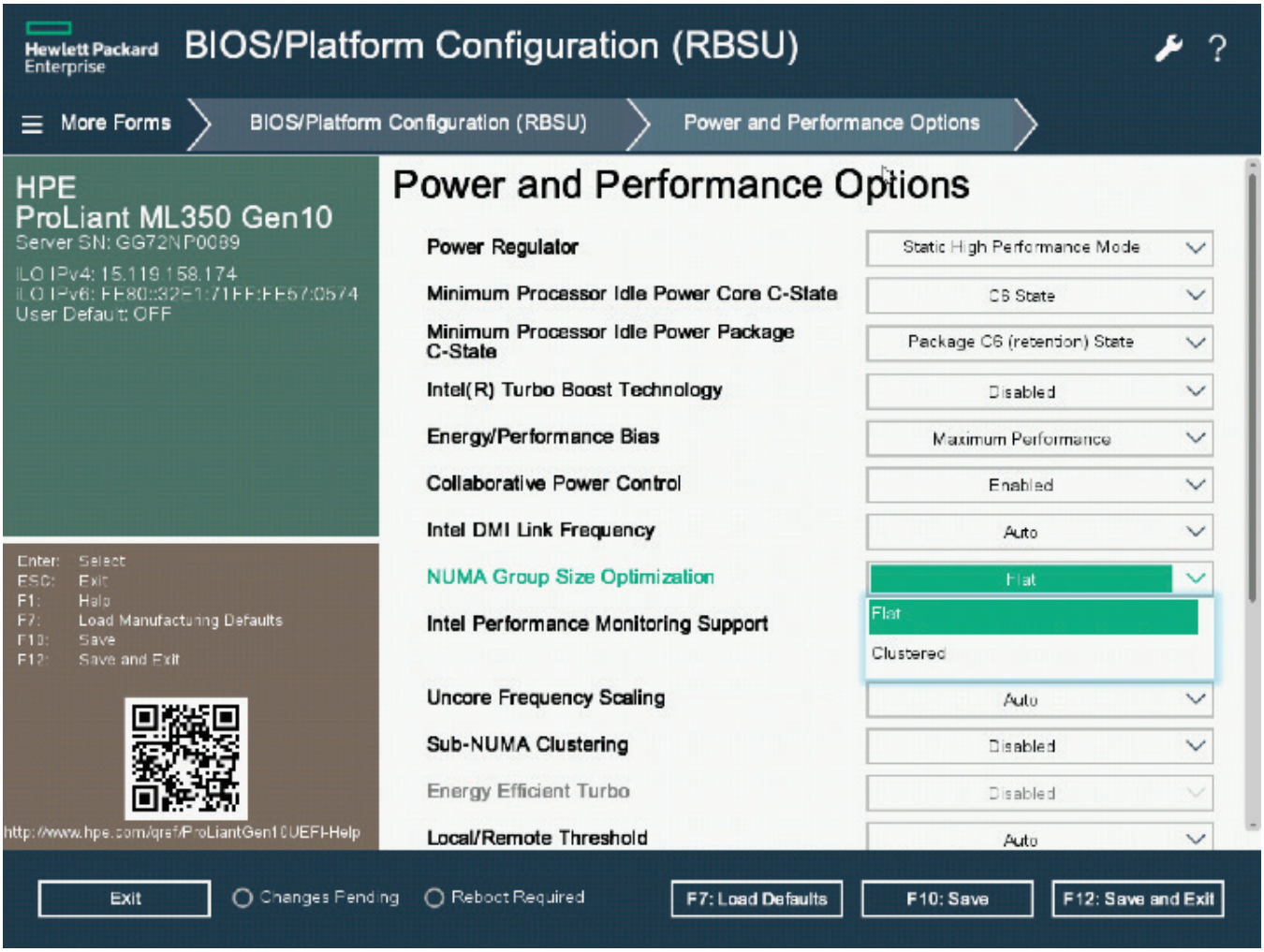
  HPE ProLiant Servers prior to Gen10, such as Gen9, Gen8, and G7, use QPI as its interconnect technology.

  - Two QPI Links
    - Systems with processors that support two QPI links can only be ring topology in NUMA and may introduce extra latency when accessing remote memory. Intel Xeon E5-4600 v4/v3 are processors of this kind.
  - Three QPI Links
    - Systems with processors that support three QPI links can be fully connected in NUMA. Intel Xeon E7 4800/8800 v4/v3 are processors of this kind.

New NUMA features on HPE ProLiant Gen10 Servers

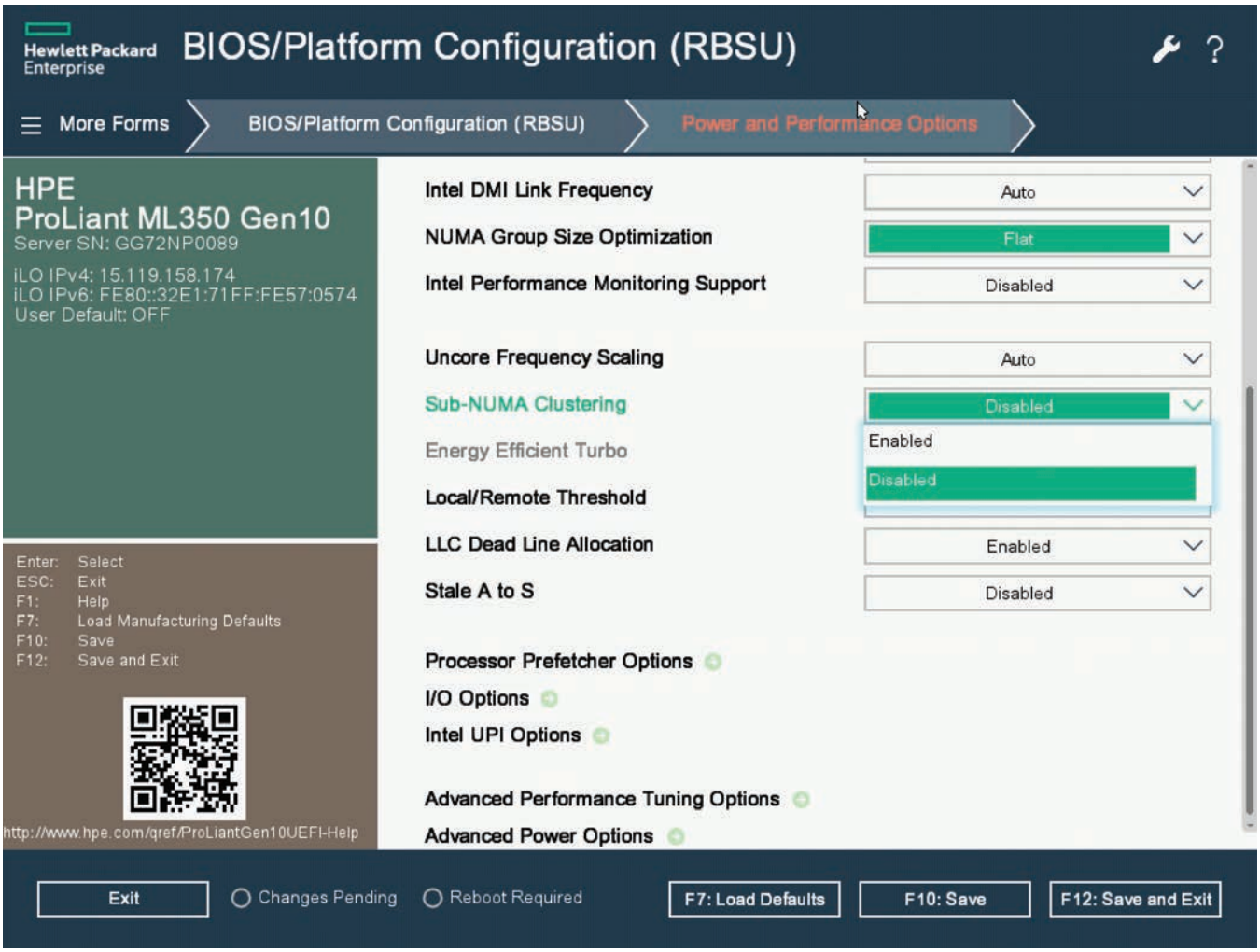
NUMA group size optimization

Use this option to configure how the System BIOS reports the size of a NUMA node (number of logical processors), which assists the operating system in grouping processors for application use (referred to as Kgroups). The default setting of Clustered provides better performance due to optimizing the resulting groups along the NUMA boundaries. However, some application might not be optimized to take advantage of processors spanning multiple groups. In such cases, selecting the Flat option might be necessary for those applications to utilize more logical processors.



Sub-NUMA cluster on Intel Xeon Scalable processor family

When enabled, sub-NUMA clustering divides the processor's cores, cache, and memory into multiple NUMA domains. Enabling this feature can increase performance for workloads that are NUMA aware and optimized. Note that when this option is enabled, up to 1 GB of system memory may become unavailable.



Two-node servers

Description

Figure 2 shows a simple schematic diagram of two-node servers with two Intel multi-core processors. Each processor with its associated memory is a Linux NUMA node and each contains four, six, eight, or more processor cores, depending upon the model. The bold line connecting the nodes in the diagram represents an interconnect link, such as UPI or QPI. The end result is a simple NUMA topology in which each node is one "hop" away from the other. Remote one-hop references to memory are performed through the interconnect link and exhibit higher latencies and proportionally lower bandwidth than performing memory references on the local node.

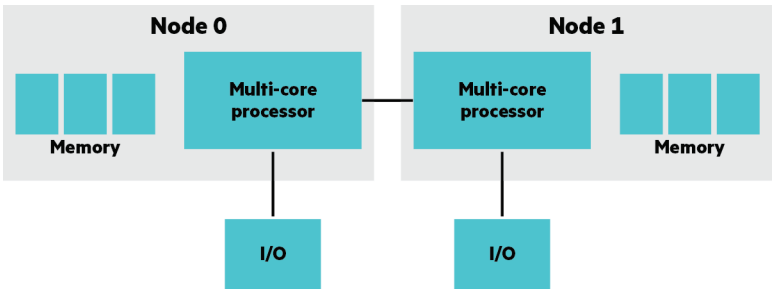


Figure 2. HPE ProLiant two-node x86-64 NUMA server

List of HPE ProLiant two-node servers

- HPE ProLiant DL360 Gen10 2P and Gen9 2P
- HPE ProLiant DL380 Gen10 2P and Gen9 2P
- HPE ProLiant ML350 Gen10 2P and Gen9 2P
- HPE Synergy 480 Gen10 and Gen9
- HPE Synergy 620 Gen9

Obtaining information about two-node server hardware resources and NUMA topology—Using HPE ProLiant ML350 Gen10 as an example

To obtain information about the hardware resources and the NUMA topology of a Linux system, you can use the `numactl` command. The `--hardware` option in the following example requests `numactl` to display CPUs, total memory, and a snapshot of free memory on each node. The Linux kernel's logical node IDs shown in the following `numactl` output correspond to the node numbers in Figure 2:

```
ML350> numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 20 21 22 23 24 25 26 27 28 29
node 0 size: 7852 MB
node 0 free: 6372 MB
node 1 cpus: 10 11 12 13 14 15 16 17 18 19 30 31 32 33 34 35 36 37 38 39
node 1 size: 8191 MB
node 1 free: 6902 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

The nodes on this system contain 10 processor cores each, which Linux identifies as CPUs. Processor hyper-threading is enabled on this system. The number of cores on each node will be half when hyper-threading is disabled.

The table under the node distances line represents the relative distance or latency between the node listed in the row heading and the node listed in the column heading. These values are typically provided by the system firmware in a table called the system locality information table (SLIT). The SLIT is one of the system description tables provided by the advanced configuration and power interface (ACPI) implementation in system firmware. For more information about SLIT data, refer to the "Avoiding node memory overflow resulting in off-node allocations" section. For more information about ACPI, refer to the ACPI webpage at acpi.info.

Note
The acpi.info and other links might take you outside the HPE website. We do not control and are not responsible for information outside of hpe.com.

Analyzing two-node server's NUMA topology and internode bandwidth

Not all HPE ProLiant x86-64 NUMA servers have a populated SLIT. The HPE ProLiant ML350 Gen10 Server is an example. When you see a node distance table containing all 10s and 20s, as shown in the preceding example, this usually means the system's firmware did not supply SLIT values and the kernel constructed a default table for node distances. By convention, the distance value for accesses to the local node is 10, whether supplied by a SLIT or by the kernel when it creates a default SLIT. In the node distances table in the preceding example, entries on the diagonal from top-left to bottom-right represent the local accesses. The kernel's default distance value for all remote nodes is 20. So, by default, the kernel assumes that remote accesses are twice as expensive (or far away) as local accesses and that no node is more than one hop away from another. This is a close approximation for HPE ProLiant ML350 Gen10 Server, as shown in the following internode bandwidth table.

Table 1 shows the internode bandwidth of this HPE ProLiant ML350 Gen10 Server with 2.40 GHz 10-core Intel Xeon Gold 5115 Processors for memory loads and stores (combined read/write workload), as performed with the `memcpy` function.

Each row in the table shows the results of a test run on one of the indicated node's CPUs (cores). Each column represents the node where the test's memory area was allocated.

Table 1. HPE ProLiant ML350 Gen10 internode read/write bandwidth (memcpy)

	Memory on	
Task on	**Node 0**	**Node 1**
Node 0	1.00	0.58
Node 1	0.59	1.00

To show the relative cost of off-node references, the values in the table are normalized to the average local bandwidth of all the nodes.

The memory bandwidth measured for remote accesses is 58 percent of the local bandwidth, so the penalty for accessing off-node memory is significant. In fact, accessing memory on a remote node is nearly twice the cost of accessing it on a local node; therefore, the measured values correspond relatively well to the kernel's default SLIT distance values.

Techniques for avoiding or minimizing this remote access penalty will be described later in this paper. In particular, the following "Constraining task migration to improve locality" section discusses how to bind tasks to a set of CPUs to constrain memory locality.

Four-socket servers—Ring topology

Description

Figure 3 shows a simple schematic diagram of a four-socket server with ring topology, with its four multi-core Intel processors. Each processor and its associated memory is a Linux NUMA node and each processor contains four, six, eight or more processor cores, depending upon the model.

In this NUMA topology, its nodes are connected in a ring by interconnect links, such as UPI or QPI, with each node connected to two other nodes only. Consequently, any given node is one hop away from two other nodes, and two hops away from the remaining node in the system.

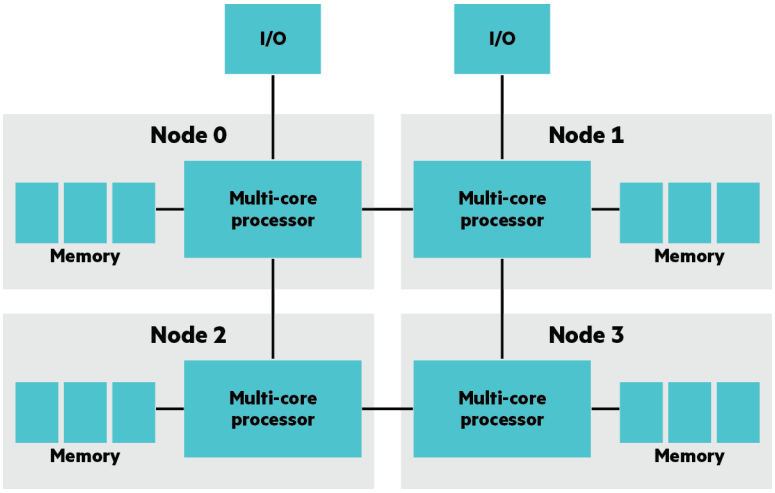


Figure 3. HPE ProLiant four-socket x86-64 NUMA server

List of HPE four-node ring topology servers

- HPE ProLiant DL560 Gen10 with Intel Xeon Gold Edition 5100 Processor series (two UPI links)
- HPE ProLiant DL580 Gen10 with Intel Xeon Gold Edition 5100 Processor series (two UPI links)
- HPE Synergy 660 Gen10 with Intel Xeon Scalable processors Gold Edition 5100 series (two UPI links)
- HPE ProLiant DL560 Gen9 with Intel Xeon E5-4600 v4/v3 Processors
- HPE ProLiant DL580 Gen9 with Intel Xeon E5-4600 v4/v3 Processors
- HPE ProLiant DL560 Gen8

Obtaining information about four-node ring topology, HPE server hardware resources, and NUMA topology—Using HPE ProLiant DL560 Gen10 as an example

The following `numactl --hardware` command output reflects the Linux kernel view of the HPE ProLiant DL560 Gen10 Server topology:

```
DL560> numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 40 41 42 43 44 45 46 47 48 49
node 0 size: 7851 MB
node 0 free: 6950 MB
node 1 cpus: 10 11 12 13 14 15 16 17 18 19 50 51 52 53 54 55 56 57 58 59
node 1 size: 8192 MB
node 1 free: 6591 MB
node 2 cpus: 20 21 22 23 24 25 26 27 28 29 60 61 62 63 64 65 66 67 68 69
node 2 size: 8192 MB
node 2 free: 7367 MB
node 3 cpus: 30 31 32 33 34 35 36 37 38 39 70 71 72 73 74 75 76 77 78 79
node 3 size: 8191 MB
node 3 free: 7268 MB
```

```
node distances:
node   0   1   2   3
  0:  10  21  21  21
  1:  21  10  21  21
  2:  21  21  10  21
  3:  21  21  21  10
```

This system has 10 cores or 20 threads of CPUs per node, with hyper-threading enabled.

Analyzing HPE ProLiant DL560 Gen10 NUMA topology and internode bandwidth

The HPE ProLiant DL560 Gen10 firmware does provide a populated SLIT in its default configuration. As shown in the preceding example, the Linux kernel uses the SLIT table to form the node distance table. The distances reflect the expected topology based on the HPE ProLiant DL560 Gen10 architecture. Each processor is local to itself, with the default distance of 10.

Table 2 shows the normalized internode bandwidth of an HPE ProLiant DL560 Gen10 Server with 2.4 GHz 10-core Intel Xeon Gold 5115 Processors for memory loads and stores (combined read/write workload) using the `memcpy` function. Each row in the table lists test results for each node; the test was run on one of the node's CPUs (cores). Each column represents the node where the test's memory area was allocated.

Table 2. HPE ProLiant DL560 Gen10 internode read/write bandwidth (`memcpy`)

	Memory			
Task on	**Node 0**	**Node 1**	**Node 2**	**Node 3**
Node 0	1.0	0.52	0.51	0.34
Node 1	0.53	1.0	0.33	0.52
Node 2	0.51	0.34	1.0	0.52
Node 3	0.34	0.52	0.53	1.0

To show the relative cost of off-node references, the values in the table are normalized to the average local bandwidth of all the nodes.

The memory bandwidth measured for "one-hop" remote accesses was 51–53 percent of the local bandwidth. However, the bandwidth measured for "two-hop" remote access was 33–34 percent of the local accesses, which is higher than "one-hop" remote accesses. So the penalty for accessing "one-hop" off-node memory is significant, but the penalty for access "two-hop" off-node memory is even higher. Programs running on the ring-topology system must be carefully designed or deployed to avoid the extra penalty when accessing memory on the indirect-connected nodes.

Some variations exist in the ratios of "local bandwidth" to "average local bandwidth" and "remote bandwidth" to "average local bandwidth," depending on the node where the task accessing the memory is running. For any given test run, some of this variation may be within the run-to-run variation. However, the pattern is repeatable enough to represent the actual server behavior.

Four-socket servers—Fully connected topology

Description

Figure 4 shows a simple schematic diagram of a four-socket server with a fully connected topology, consisting of four multi-core Intel processors. Each processor and its associated memory is a Linux NUMA node, and each processor contains six, 10, 12, or more processor cores, depending upon the model. All the nodes are connected by interconnect links, such as UPI or QPI, in a fully connected topology where each node is directly connected to all the other nodes. As a result, any given node is one "hop" away from every other node.

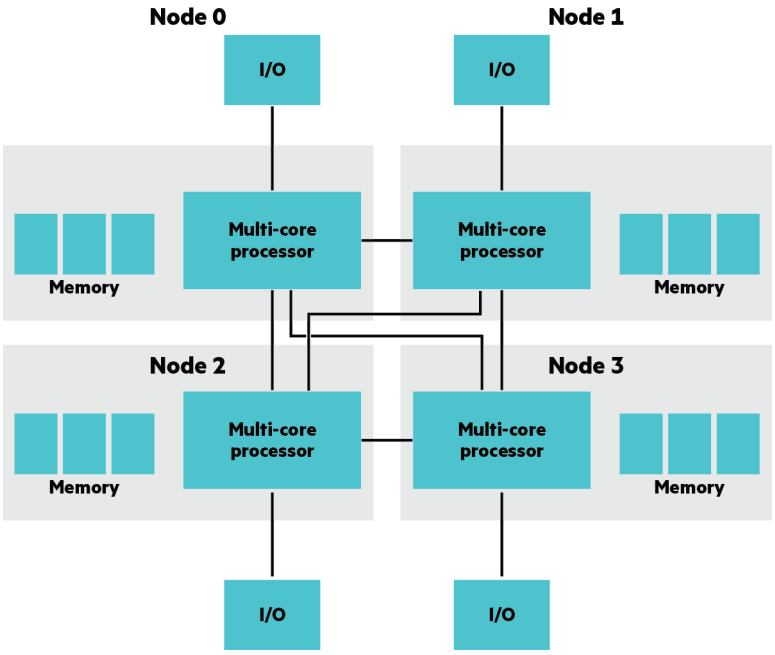


Figure 4. HPE ProLiant DL580 Gen10 four-socket x86-64 NUMA server

List of the HPE four-socket servers with fully connected topology

- HPE ProLiant DL560 Gen10 with Intel Xeon Platinum Edition or Gold 6100 Processor series (3 UPI links)
- HPE ProLiant DL580 Gen10 with Intel Xeon Platinum Edition or Gold 6100 Processor series (3 UPI links)
- HPE ProLiant DL560 Gen9 with Intel Xeon E7-4800/8800 v4/v3 processors
- HPE ProLiant DL580 Gen9 with Intel Xeon E7-4800/8800 v4/v3 processors
- HPE ProLiant DL580 G7

Obtaining information about four-socket server hardware resources and NUMA topology—Using HPE ProLiant DL580 Gen10 as an example

As in the case of the HPE ProLiant DL560 Gen10 Server, the Linux kernel's view of the server's topology using the `numactl` command is informative. Again, the `--hardware` option requests numactl to display CPUs, total memory, and a snapshot of free memory for each node:

```
DL580> numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 80 81 82 83 84 85 86 87 88 89 90 91 92
93 94 95 96 97 98 99
node 0 size: 7641 MB
node 0 free: 6574 MB
node 1 cpus: 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 100 101 102 103 104 105 106
107 108 109 110 111 112 113 114 115 116 117 118 119
node 1 size: 8058 MB
node 1 free: 7682 MB
node 2 cpus: 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 120 121 122 123 124 125 126
127 128 129 130 131 132 133 134 135 136 137 138 139
node 2 size: 8058 MB
```

```
node 2 free: 7590 MB
node 3 cpus: 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 140 141 142 143 144 145 146
147 148 149 150 151 152 153 154 155 156 157 158 159
node 3 size: 8057 MB
node 3 free: 7633 MB
node distances:
node   0   1   2   3
  0:  10  21  21  21
  1:  21  10  21  21
  2:  21  21  10  21
  3:  21  21  21  10
```

This system has 20 cores of CPUs per node, with hyper-threading enabled.

Analyzing the HPE ProLiant DL580 Gen10 NUMA topology and internode bandwidth

The HPE ProLiant DL580 Gen10 firmware does provide a populated SLIT in its default configuration. This accurately reflects the system's topology, given that all its nodes are fully connected. Furthermore, measurements of the system's behavior confirm this fact, as seen in Table 3.

Table 3 shows the internode bandwidth of HPE ProLiant DL580 Gen10 Server with 2.0 GHz 40-core Intel Xeon Gold 6138T Processors for memory loads and stores (combined read/write workload) using the `memcpy` function. Each row in the table lists test results for each node; the test was run on one of the node's CPUs (cores). Each column represents the node where the test's memory area was allocated.

Table 3. HPE ProLiant DL580 Gen10 internode read/write bandwidth (`memcpy` function)

	Memory on			
Task on	**Node 0**	**Node 1**	**Node 2**	**Node 3**
Node 0	1.0	0.47	0.46	0.44
Node 1	0.47	1.00	0.44	0.47
Node 2	0.46	0.44	1.0	0.470.59
Node 3	0.44	0.47	0.47	1.0

To show the relative cost of off-node references, the values in the table are normalized to the average local bandwidth of all the nodes. The table shows minor variations in these bandwidth measurements, depending on the node where the task accessing the memory is running. For any given test run, some of this variation might be within the run-to-run variation. However, the pattern is repeatable enough to represent the actual server behavior.

The table clearly shows that the remote access bandwidth for all nodes is roughly 44–47 percent of the local bandwidth for mixed reads and writes. The uniformity of these results agrees with the system's physical fully connected topology. Unlike the performance measured on the ring-topology system, the latency for off-node memory are equal to from one node to any other nodes.

ACPI root bridge PXM preferences option

In addition to memory locality, the HPE ProLiant DL580 Gen10 Server can optionally provide proximity information for each I/O controller by enabling PCI root bridge `device_PXM` objects, which can be used by interface drivers to optimize interrupt assignment and allocation of receive/transmit buffers.

By default, the **ACPI Root Bridge PXM Preferences** option is enabled. To change this setting, use the **BIOS/Platform Configuration (RBSU) Advanced Options** menu in BIOS. The **ACPI Root Bridge PXM Preferences** option is located under the **Advanced System ROM Options** sub-menu.

Eight-socket servers—HPE PREMA architecture

Description

Figure 5 shows a simple schematic diagram of an HPE ProLiant DL980 G7 Server consisting of eight multi-core Intel processors. Each processor and its associated memory is a Linux NUMA node, and each processor contains four, six, eight, or more processor cores, depending on the model. The processors are paired within a QPI island, where each processor in the pair is connected by QPI links with the corresponding processor of the pair and with an I/O hub (except one processor pair). Additionally, each processor is connected via QPI links to two node controllers, derived from technology that powers the HPE Integrity Superdome 2.

The four-node controllers of the HPE ProLiant DL980 G7 Server are connected in pairs using six QPI links per pair. As a result, memory bandwidth is:

- Fastest from a processor to directly attached memory (zero hops)
- A single QPI link (hop) to memory connected to the other processor in a QPI island
- Two QPI links from a QPI island connected to the same node controller (the two islands are grouped as a quad)
- Three QPI links from memory connected to any given processor in the other quad

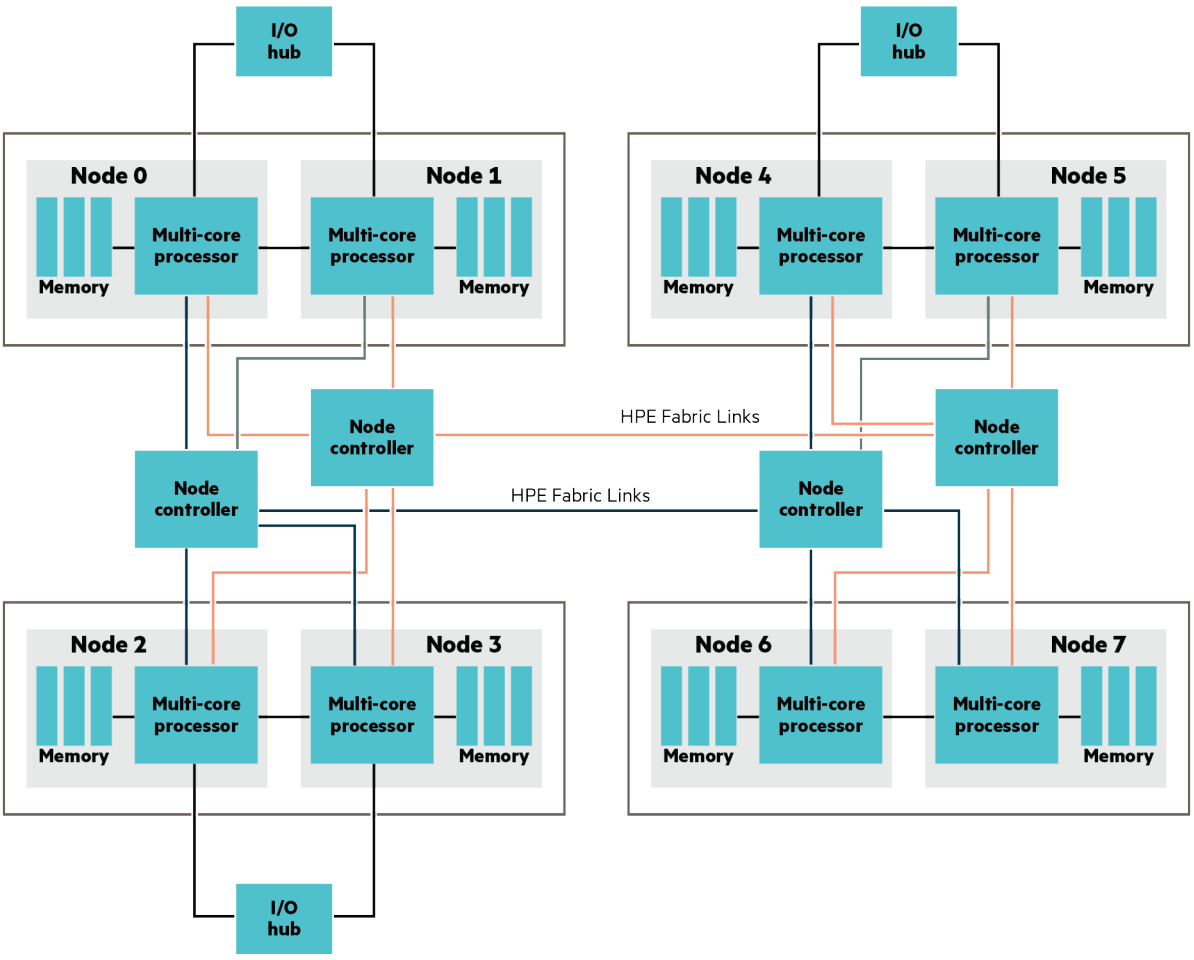


Figure 5. HPE ProLiant DL980 G7 eight-node x86-64 NUMA server

The HPE ProLiant DL980 G7 topology is an example of the HPE PREMA Architecture, where the node controllers are core components. The node controllers provide additional QPI bandwidth for hops beyond a given quad, as compared to a "glueless" eight-node topology. In a glueless topology (as shown in Figure 6), each processor connects to I/O hubs (IOHs) or other processors via one of four QPI links per processor. One QPI

link per processor would be required for an IOH connection, another for the processor in a pair or QPI island, with the remaining two processors connecting across QPI islands. To access memory between processor pairs that are not directly connected, a request must be routed via another processor (such as a request from the upper-left processor to a processor in the upper-right QPI island).

In the HPE PREMA architecture used by the HPE ProLiant DL980 G7 Server, these connections are managed by the node controllers. The node controllers provide six QPI links between quads (as opposed to four QPI links in a glueless topology); they also work with the processor cache coherency mechanism. On a cache line read request, a glueless topology requires all processors to check for the requested memory line and provide the most up-to-date version (as applicable). The HPE PREMA architecture node controllers participate in the cache coherency mechanism of the HPE ProLiant DL980 G7 Server by tracking when a QPI island has a copy of a memory line. So if a cache line read request occurs, the coherency check (snoop or snoop packet) is issued to the other processor in the QPI island and to a node controller, which can respond for the remote QPI islands. This process reduces the bandwidth required to support cache coherence by:

(1) Not sending unnecessary requests when no other QPI island has the memory line cached

(2) By directing the snoop packet to the specific QPI island that has the memory line copy in its cache, instead of to the entire system

In addition to the performance benefits of the additional QPI bandwidth (when compared to the glueless topology), the HPE ProLiant DL980 G7 topology also provides additional datapath redundancy. Bandwidth across the node controller fabric is dynamically balanced, allowing failing or problematic QPI links between node controllers to be rerouted, as well as reducing downtime.

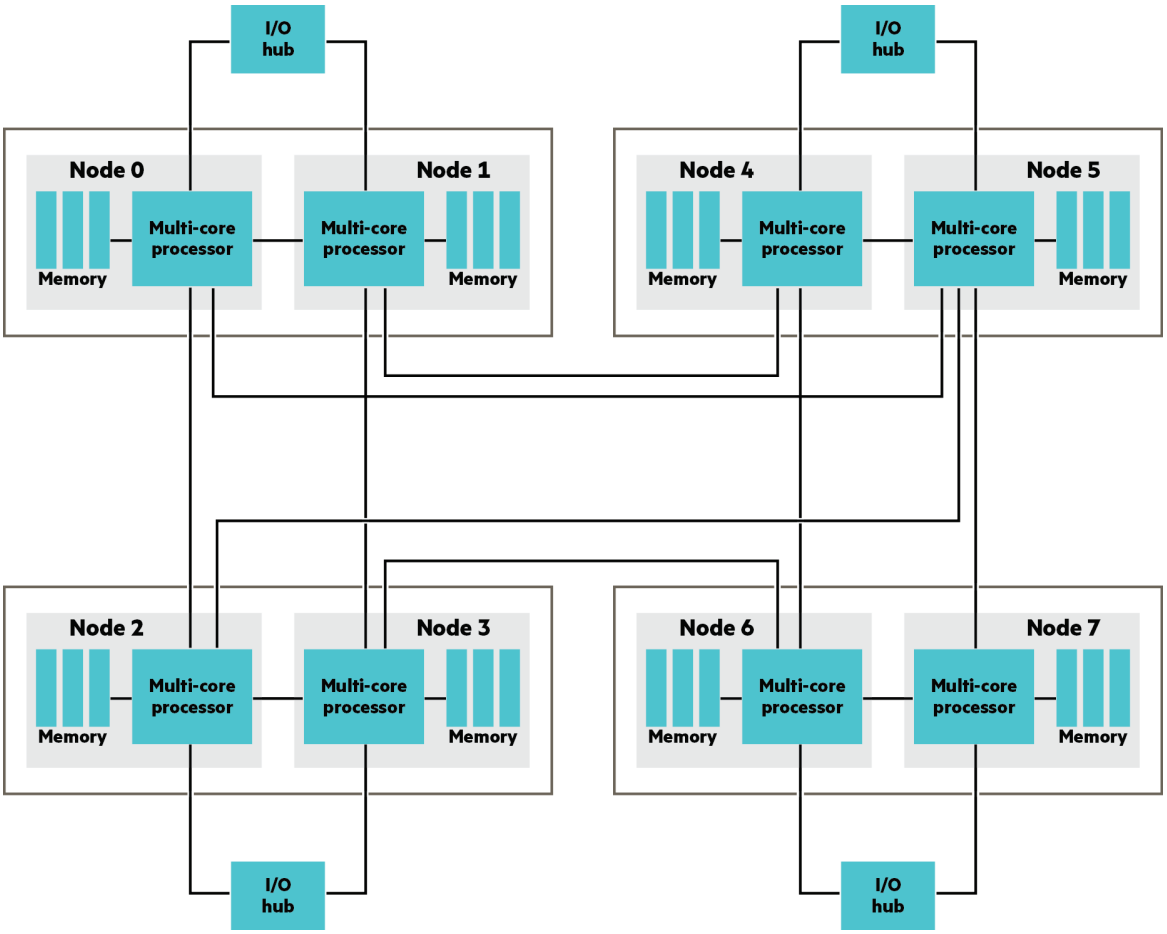


Figure 6. Generic eight-socket glueless topology[2]

[2] This is a representation of a generic eight-socket glueless topology. For the HPE ProLiant DL980 architecture, see Figure 5.

HPE eight-node server with HPE PREMA architecture

- HPE ProLiant DL980 G7

Obtaining information about eight-node server hardware resources and NUMA topology—Using HPE ProLiant DL980 G7 as an example

The following `numactl --hardware` command output reflects the Red Hat Enterprise Linux kernel view of the HPE ProLiant DL980 G7 topology:

```
DL980> numactl --hardware
available: 8 nodes (0-7)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9
node 0 size: 131061 MB
node 0 free: 108211 MB
node 1 cpus: 10 11 12 13 14 15 16 17 18 19
node 1 size: 131072 MB
node 1 free: 108245 MB
node 2 cpus: 20 21 22 23 24 25 26 27 28 29
node 2 size: 131072 MB
node 2 free: 108267 MB
node 3 cpus: 30 31 32 33 34 35 36 37 38 39
node 3 size: 131072 MB
node 3 free: 108068 MB
node 4 cpus: 40 41 42 43 44 45 46 47 48 49
node 4 size: 131072 MB
node 4 free: 108355 MB
node 5 cpus: 50 51 52 53 54 55 56 57 58 59
node 5 size: 131072 MB
node 5 free: 108295 MB
node 6 cpus: 60 61 62 63 64 65 66 67 68 69
node 6 size: 131072 MB
node 6 free: 108361 MB
node 7 cpus: 70 71 72 73 74 75 76 77 78 79
node 7 size: 131072 MB
node 7 free: 107574 MB
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  12  17  17  19  19  19  19
  1:  12  10  17  17  19  19  19  19
  2:  17  17  10  12  19  19  19  19
  3:  17  17  12  10  19  19  19  19
  4:  19  19  19  19  10  12  17  17
  5:  19  19  19  19  12  10  17  17
  6:  19  19  19  19  17  17  10  12
  7:  19  19  19  19  17  17  12  10
```

The system has 10 cores per node and is not hyper-threaded. A hyper-threaded example is omitted for brevity but would add cores starting with CPU IDs 80 to 89 (to node 0) proceeding by tens in order.

Analyzing the HPE ProLiant DL980 G7 NUMA topology and internode bandwidth

The HPE ProLiant DL980 G7 firmware does provide a populated SLIT in its default configuration. As shown in the preceding example, the Red Hat Enterprise Linux kernel uses the SLIT table to form the node distance table. The distances reflect the expected topology based on the HPE ProLiant DL980 G7 and HPE PREMA architecture. Each processor is local to itself (the default distance of 10).

Table 4 shows the normalized memory bandwidth for a task executing in a given node and accessing memory in another given node.

Table 4. HPE ProLiant DL980 G7 internode read/write bandwidth (`memcpy` function)

Node	0	1	2	3	4	5	6	7
0	1.00	0.85	0.66	0.66	0.62	0.62	0.62	0.62
1	0.85	1.00	0.67	0.67	0.63	0.63	0.63	0.62
2	0.67	0.67	1.00	0.86	0.62	0.63	0.63	0.63
3	0.66	0.66	0.85	1.00	0.61	0.61	0.62	0.61
4	0.62	0.62	0.62	0.62	1.00	0.85	0.67	0.66
5	0.62	0.63	0.63	0.62	0.86	1.00	0.68	0.70
6	0.59	0.59	0.59	0.59	0.63	0.66	1.00	0.84
7	0.63	0.64	0.64	0.64	0.68	0.67	0.87	1.00

For comparison, Table 5 normalizes the SLIT table given to the Red Hat Enterprise Linux kernel by the HPE ProLiant DL980 G7 Server.

Note

The SLIT table, while a trifle pessimistic, is a fairly accurate summation of the relative cost of access between a given pair of nodes. The Red Hat Enterprise Linux kernel can then use the SLIT table to properly weigh the costs of access, migration across processors (or nodes), and other activities for a given task. Advanced applications or programmers can use this information to manually place tasks and/or memory based on the known pattern of access for the application.

Table 5. HPE ProLiant DL980 G7 normalized SLIT table

Node	0	1	2	3	4	5	6	7
0	1.00	0.83	0.59	0.59	0.53	0.53	0.53	0.53
1	0.83	1.00	0.59	0.59	0.53	0.53	0.53	0.53
2	0.59	0.59	1.00	0.83	0.53	0.53	0.53	0.53
3	0.59	0.59	0.83	1.00	0.53	0.53	0.53	0.53
4	0.53	0.53	0.53	0.53	1.00	0.83	0.59	0.59
5	0.53	0.53	0.53	0.53	0.83	1.00	0.59	0.59
6	0.53	0.53	0.53	0.53	0.59	0.59	1.00	0.83
7	0.53	0.53	0.53	0.53	0.59	0.59	0.83	1.00

Firmware-configured node interleaving on HPE ProLiant servers

On many HPE ProLiant servers, the RBSU allows enabling or disabling a configuration option called node interleaving. (This option is accessible through the system firmware's graphical console interface when the server is powered up or reset.) This option is disabled by default. When the option is enabled, the system firmware configures the hardware to interleave sequential physical page addresses over all the nodes/sockets in the system. Upon boot, the system firmware informs the kernel that the server is a uniform memory access system, with all memory in one large pool (possibly containing holes that can be ignored in this context).

For older versions of the Red Hat Enterprise Linux kernel, such as found in Red Hat Enterprise Linux 4, experience has shown that some workloads perform better or more predictably when node interleaving is enabled. As a result, some application installation guides—and even some certification reports—recommend or require that node interleaving be enabled.

However, with improved Linux NUMA support in newer versions of the kernel, as found in Red Hat Enterprise Linux 5, 6, and 7, node interleaving is seldom advantageous, especially on larger system configurations. In some cases, node interleaving can lead to significant performance degradation. When node interleaving is enabled in the system's firmware, the kernel has no knowledge of the location of memory pages with respect to the system's actual NUMA topology. This effectively disables the kernel's NUMA optimizations. As a result, data structures can be allocated on pages that are maximally remote from where they are heavily used, leading to suboptimal kernel and application performance. However, even with the newer kernels, some application guides still recommend enabling firmware node interleaving, and some administrators and support personnel enable it or recommend it out of habit.

Important

Unless you have an application that will not work or will not be supported by the vendor without firmware node interleaving enabled, for best performance, we recommend that you do not enable firmware node interleaving. Especially, on HPE ProLiant running Red Hat Enterprise Linux 5, 6, or 7. First, try to optimize your system with the tuning methods discussed in the remainder of this document. These methods include the use of interleaved memory allocation policies that can be applied to individual applications or groups of applications where necessary, without adversely affecting the kernel or the rest of the system—a technique that is generally preferable to firmware node interleaving.

Red Hat Enterprise Linux memory policy principle

This section introduces the principles underlying the Red Hat Enterprise Linux kernel's NUMA memory management features. Examples introduce the commands available to query and control the behavior of these features. This section also explains how and when these features should be used to maximize locality, minimize loading on the system interconnects, and boost application performance.

Local allocation of memory and kernel data structures

Linux treats each NUMA node that contains memory as a separately managed pool of available memory. Each pool has its own list of free pages, "least recently used" (LRU) lists to manage in-use pages, statistics, and other management structures including the locks that serialize access to the lists. The kernel allocates these data structures in the memory of the node managed by those data structures. Linux maximizes the locality of its own memory accesses and that of the tasks requesting the memory by causing requests for memory from CPUs local to a node to favor that node's page pool.

In addition to the effective memory bandwidth benefits of locality, the use of per-node locks—which are accessed more efficiently by CPUs local to the node—improves the scalability and throughput of the page allocator. For example, an HPE ProLiant DL580 Gen10 with 256 GB of memory has more than 64 million 4 KB pages to manage. If node interleaving is enabled by the server's firmware (which is not recommended), these pages all appear as a single page pool protected by a single lock. That lock resides in one node's memory—the server's physical NUMA characteristics still apply—and is remote from CPUs on all other nodes.

With node interleaving disabled on a four-socket HPE ProLiant DL580, Linux manages the pages in four pools of 16 million pages each. That might be a large number of pages to manage, but the disabled interleaving improves scalability. Red Hat Enterprise Linux's NUMA features result in a smaller set of CPUs—those local to the node—accessing the node's lock and these CPUs access the local data structures they protect much more often than remote CPUs do.

Red Hat Enterprise Linux defaults to the local allocation policy

The mechanism that Red Hat Enterprise Linux uses to direct memory requests to the page allocator of the local node is called the local allocation policy, which is the default memory allocation policy for all tasks in the system after startup. All tasks' memory policies continue to default to the system default policy unless explicitly changed by an administrator, a user, or a NUMA-aware application. The default local allocation policy is

sufficient for many tasks. However, if your application has unique requirements or does not perform to your expectations, you can explore other Linux-supported memory allocation policies discussed in subsequent sections of this paper.

Local allocation provides the memory with optimum locality as long as the user of that memory continues to execute on CPUs local to the node. The Linux scheduler tends to keep tasks executing on CPUs on or close to the node where they started, but periods of load imbalance can result in tasks—especially long-lived ones—being migrated away from their original memory allocation. You can constrain task migrations to improve locality, as discussed in the "Constraining task migration to improve locality" section of this document.

Local allocation policy always attempts to allocate a page from the node local to the CPU where the requesting task executes; however, what happens when the local node has no available free pages? The local allocation policy is forgiving in this respect. If an attempted local allocation fails for lack of local memory, the attempt falls back (overflows) to another node in the system, preferably to the closest nearby node. The "Avoiding node memory overflow resulting in off-node allocations" section discusses what happens when node overflow/fallback occurs and the implications of that occurrence.

Sometimes an administrator prefers that the system attempt to free up some local pages before overflowing the allocation to another node. The "Using the `vm.zone_reclaim_mode`" section describes a tunable kernel parameter to control this behavior.

Constraining task migration to improve locality

To limit the range of task migrations, you can bind or affinitize tasks to a set of CPUs. To achieve this, use the `taskset` command to set a task's CPU affinity when launching a command:

```
taskset --cpu-list <cpus-in-some-node(s)> <some-command>
```

For a previously started task, if you know its process ID (pid), you can use this command:

```
taskset -p --cpu-list <cpus-in-some-node(s)> <process-id>
```

Changing a running task's CPU affinity using the `taskset -p` command affects only the locality of memory allocated after the change in affinity. **Memory allocation policy applies only at page allocation time**. The "Controlling page migration" section of this document discusses a mechanism for migrating task memory between nodes once the pages have been allocated. For more information about the taskset command, see the `taskset(1)` manpage.

The `numactl` command allows you to specify the CPU affinity by node ID in addition to CPU IDs, but only when launching the program:

```
numactl --cpunodebind<some-node(s)> <some-command>
```

For example, if you want to start a database server on two nodes of an HPE ProLiant DL580 Gen10 Server to minimize the distance from the database server's allocated memory to its storage devices, you could specify:

```
numactl --cpunodebind 2,3 startdb <db-args>
```

In all these cases, the indicated tasks are bound to the specified set of CPUs. If all these tasks use the local allocation policy, any future memory allocations come from one of the specified nodes, because the task(s) are constrained to run on CPUs local to those nodes. Previously allocated pages remain where they were allocated.

Important

For this reason, you can achieve the best locality by setting an application's CPU affinity at fork time, as with the first form of taskset or using `numactl --cpunodebind`. Doing so guarantees that all kernel data structures associated with the applications' component tasks, and all the tasks' private data pages, are located on or near the nodes containing the specified CPUs. It also warrants that the tasks will not migrate too far from where their memory was allocated.

For information about another Linux feature that provides a mechanism for constraining task migrations and memory allocations, see the "Using cpusets (privileged users only) to Group Tasks and Constrain CPU/Memory Usage" section of this document. Indeed, all the pages mapped by the task—including shared pages mapped by more than one task—are migrated to node 2.

How to Use NUMA maps (numa_maps) to determine where task pages are allocated

Before discussing additional memory policies and how and when to use them, this section first explains how to determine what memory policies are being used by a task and where the kernel has allocated the task's pages. Linux exposes many details about a task's state through the /proc file system. To examine a task's memory allocation policies and resulting page placement, you can examine the /proc/<pid>/numa_maps file. This step is often done to identify potential page placement problems on a NUMA system when an application is not performing well. For example, the following command displays the shell's numa_maps:

```
cat /proc/$$/numa_maps
```

The shell parameter $$ evaluates to the shell's process ID. For a bash shell on Red Hat Enterprise Linux 7, this displays more than 30 lines of information. This discussion considers only those lines containing the text bash, stack, heap, and libc:

```
00400000 default file=/usr/bin/bash mapped=173 mapmax=4 N0=143 N1=30 kernelpagesize_kB=4
006dc000 default file=/usr/bin/bash anon=1 dirty=1 N0=1 kernelpagesize_kB=4
006e6000 default anon=6 dirty=6 N0=6 kernelpagesize_kB=4
010dd000 default heap anon=345 dirty=345 N0=223 N1=122 kernelpagesize_kB=4
7f848efe5000 default file=/usr/lib64/libc-2.17.so mapped=186 mapmax=109 N1=186 kernelpagesize_kB=4
7fffaa3be000 default stack anon=8 dirty=8 N0=6 N1=2 kernelpagesize_kB=4
```

Each line in the numa_maps file represents one segment or region of the task's virtual address space. Linux calls these ranges of virtual addresses virtual memory areas (VMAs). The first column shows the start address of the VMA. The end address and additional details—such as protections, file offset, and so forth—can be obtained from a parallel file named /proc/<pid>/maps (no "numa_" in the file name). Normally, you use these two files together to understand the layout of a task, using the start address as the key to match the lines of the two files.

The second column shows the memory allocation policy for that range of virtual memory. The value default indicates that neither this range of virtual memory nor the task itself has an assigned memory allocation policy, so both are using the system default policy. As discussed earlier, the system uses the local allocation policy by default.

The third column specifies the path of a mapped file, or alternatively, a special segment name such as heap or stack.

Note
Portions of the file /bin/bash—the bash executable image—are mapped at three different addresses. Examination of the task's maps file would reveal the offsets, ranges, and protections of these three segments. It would also distinguish the executable text section, the read-only constant data section, and the read-write initialized data section.

The remaining fields provide information about the state and location of any physical memory pages backing the VMA.

Important
The numa_maps file shows only those pages that the task has accessed at least once and that still remain in the task's page tables. The page counts do not necessarily represent the entire memory usage of the VMA—especially for mapped files.

The mapped= item shows the number of pages mapped from the file. This field does not display unless its value differs from the anon= and dirty= page counts, discussed later in this document.

Note
The first libc line and the first bash line display mapped= counts, as these are all mapped file pages (not anonymous). None are dirty because those VMAs are generally not written to.

The mapmax= shows the maximum number of page table entries that reference any page in the VMA. At least one page of the shared executable text portion of libc has 139-page table references. This field will not display if the count is 1.

The anon= and dirty= items display the number of pages of the VMA in those states. Anonymous pages, such as a task's stack or data pages, are generally private to a VMA with no file backing them. Unlike file-backed pages, anonymous pages are freed when unmapped from a task. Dirty pages are pages that have been modified by the application, such that the only valid copy of the data is in the page memory.

Finally, the N<nodeid>= items show the number of pages allocated on each listed <nodeid>. Again, this only includes pages actually referenced by the task's page table. In the excerpt from the shell's numa_maps shown earlier, the heap and stack each have pages on all four nodes. This means that the shell probably migrated among CPUs on nodes 0, 1, 2, and 3 and allocated pages locally on each node during its lifetime.

Memory allocation policies

The system default local allocation policy introduced in the "Red Hat Enterprise Linux defaults to the local allocation policy" section of this document is sufficient for many applications and workloads, especially when the range of task migrations is constrained to a set of CPUs on neighboring nodes. However, some applications—especially those whose CPU and memory resource requirements exceed the resources of a single node—can sometimes benefit from other memory allocation policies. Linux allows you to change individual tasks' memory policies, and even the memory policy used for ranges of tasks' virtual memory space.

The following subsections describe the scope and effect of the various memory policies supported by Linux. Generally, these policies are most effective for designing or enhancing applications to be NUMA-aware. However, some policies might be applied to existing binary applications through command line tools or inheritance.

Using Memory Policy Scope to Control Allocation

If a task can have per-task memory policies and even per-address range policies, what determines which policy applies to a given page allocation? The answer is a concept called, herein, **memory policy scope**. The following subsections describe various memory policies that define memory policy scope.

System default policy

As the name implies, this is the default policy—the one used when no other policy applies. The system default policy specifies local allocation.

Task memory policy

The task memory policy, when specified, is used for all memory allocations made by the task except those controlled by any of the memory policy scopes described in the remainder of this section. Task memory policy is inherited across fork() and exec() system calls, so it can be set for a NUMA-unaware application by setting the policy of the shell script or program that launches the application. All descendants of the launch shell or program inherit its memory policy.

The numactl command has several uses pertaining to the task memory policy:

- To start a program with a specified task memory allocation policy:

  ```
  numactl --membind=2 <some-program>
  ```

  The --membind=2 option specifies that all of the task's memory must come from node 2. No off-node allocations are allowed. (The bind memory policy mode is discussed in a subsequent section.) This form of the numactl command sets its own task memory policy to bind to node 2 using the set_mempolicy system call. (For more information, refer to the set_mempolicy(2) manpage.) This policy is then inherited by the specified program (<some-program>).

- To display the task memory policy of numactl itself:

  ```
  numactl --show
  ```

  In this form, numactl uses the get_mempolicy system call to query its own inherited memory policy and uses the sched_get_affinity system call to query its inherited CPU affinity mask. It then displays the results. This command is useful for several purposes. For one, you can use the numactl --show command to query the task policy of the running shell. The shell's task memory policy is inherited by any programs launched from that shell. Such a query might reveal useful information, if not already known. The following example shows the command issued on an HPE ProLiant DL580 G7 Server from a shell that has no task policy:

  ```
  DL580> 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
32 33 34 35 36 37 38 39
cpubind: 0 1 2 3
nodebind: 0 1 2 3
membind: 0 1 2 3
```

This example shows that the task policy is default. The other lines indicate the CPUs that the task might run on and the nodes from which the task might allocate memory (all four). The `cpubind` and `nodebind` lines both represent the set of nodes on whose CPUs the task might execute, by default, or as specified by the `--cpunodebind` option.

- To determine the effect of a numactl command in a trial run:

  Use `numactl --show as the <some-program>` argument to the `numactl --membind=2 <some-program>` command, as in the following example:

```
DL580> numactl --membind=2 numactl --show
policy: bind preferred node: 4
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
32 33 34 35 36 37 38 39
cpubind: 0 1 2 3
nodebind: 0 1 2 3
membind: 2
```

  This example shows that bind is now the task policy and the only node from which memory can be allocated is node 2. The preferred node does not apply to bind policy and can be ignored here.

  The following example shows how to infer a task's task memory policy from the numa_maps output:

```
DL580> numactl --preferred=2 sleep 30 &
[1] 4015
DL580> cat /proc/4015/numa_maps | egrep 'libc|bash|stack|heap'
0080e000 prefer:2 heap anon=2 dirty=2 N2=2
382d200000 prefer:2 file=/lib64/libc-2.12.so mapped=79 mapmax=114 N3=79 382d397000 prefer:2
file=/lib64/libc-2.12.so
382d597000 prefer:2 file=/lib64/libc-2.12.so anon=1 dirty=1 mapped=4 mapmax=48 N2=1 N3=3
382d59b000 prefer:2 file=/lib64/libc-2.12.so anon=1 dirty=1 N2=1 7fffc43d2000 prefer:2 stack anon=3
dirty=3 N2=3
```

All the virtual memory ranges listed in this example show the same memory policy: **`prefer:2`**. For more information about the preferred memory policy mode, refer to the "Preferred mode" section. The configuration in the example can only occur in one of two circumstances: (1) the task set the memory policy of each of its VMAs to this policy, which is unlikely (for more information, see the "VMA memory policy" section of this document).

(2) all these ranges have no memory policy specified, so they default to the task's memory policy. The task policy must be non-default; if it were default, the `numa_maps` lines would indicate so.

VMA memory policy

The VMA memory policy applies to a range of a task's virtual memory—a VMA. The policy can be applied only by the task itself, using the mbind system call or one of its library wrappers. Because the policy is tied to a task's virtual memory layout, it can be inherited across fork, which duplicates the parent's address space for the child. However, VMA memory policy cannot be inherited across one of the exec functions, as these functions destroy the caller's address space and construct a new one for the new executable image.

The lines in a task's `numa_maps` show the effective memory policy for the respective ranges of the task's address space. If a VMA memory policy has been specified for a region, it displays; otherwise, the task's memory policy displays, if any.

If neither VMA nor task memory policy exists, the line specifies the default, indicating the system default policy (local allocation). This logic illustrates the decision path that the kernel uses when deciding what policy to use to allocate a page for a given virtual address of a task's address space.

The following is an excerpt from the `numa_maps` of a test program started with `numactl --preferred=2`. The test program then sets the VMA memory policy for the 64 MB anonymous region at `0x7f31fe5af000 to preferred=3` and then allocates memory for that region:

```
00400000 prefer:2 file=/usr/local/bin/memtoy mapped=15 active=0 N0=15 00611000 prefer:2
file=/usr/local/bin/memtoy anon=1 dirty=1 N2=1 0229f000 prefer:2 heap anon=29 dirty=29 N2=29
382d200000 prefer:2 file=/lib64/libc-2.12.so mapped=150 mapmax=115 N0=3 N3=147 382d397000 prefer:2
file=/lib64/libc-2.12.so
382d597000 prefer:2 file=/lib64/libc-2.12.so anon=2 dirty=2 mapped=4 mapmax=48 N2=2 N3=2
382d59b000 prefer:2 file=/lib64/libc-2.12.so anon=1 dirty=1 N2=1 7f31fe5af000 prefer:3 anon=16384
dirty=16384 N3=16384 7fff68f2e000 prefer:2 stack anon=5 dirty=5 N2=5
```

The prevalence of the **`prefer:2`** policy indicates that the preferred mode is almost certainly the task's memory policy.

Note

Most of the pages of this program are, indeed, on node 2. However, the 15 pages of the executable text on the first line appear to be on node 0. Evidently, these pages were in the page cache from a previous run of this program and this current run used them from where they had been previously allocated. This demonstrates another reason for checking the `numa_maps` of an application to make sure the layout matches your expectations.

The second-to-last line in the output in this example shows the anonymous region (no `file=<path>` indicated) with a VMA memory policy specified: **`prefer:3`**. All 16,384 pages of this region have been allocated on node 3, as expected.

Shared memory policy

Shared memory policy scope is similar to VMA memory policy. However, rather than applying to a range of a task's virtual address space, shared memory policy applies directly to a shared object, such as a shared memory region (`shmem`). The same policy is seen and used by all tasks that map the shared object.

The following examples are excerpts from the `numa_maps` of two tasks that attach to a shared memory segment. Again, the tasks were started with a task memory policy of `preferred=2`, using `numactl`. One task created the segment, set the memory policy to `preferred=3` using `mbind`, and "touched" (wrote to) each page of the region to populate it. The other task simply attached the segment.

```
DL580> cat /proc/4552/numa_maps | egrep 'memtoy|heap|stack|SYSV'
00400000 prefer:2 file=/usr/local/bin/memtoy mapped=15 mapmax=2 active=13 N0=15 00611000 prefer:2
file=/usr/local/bin/memtoy anon=1 dirty=1 N2=1
006d3000 prefer:2 heap anon=29 dirty=29 N2=29
7fa4257b1000 prefer:3 file=/SYSV00000000\040(deleted) dirty=16384 active=0 N3=16384
7fffe2fda000 prefer:2 stack anon=6 dirty=6 N2=6
DL580> cat /proc/4559/numa_maps | egrep 'memtoy|heap|stack|SYSV'
00400000 prefer:2 file=/usr/local/bin/memtoy mapped=15 mapmax=2 active=13 N0=12 00611000 prefer:2
file=/usr/local/bin/memtoy anon=1 dirty=1 N2=1
01079000 prefer:2 heap anon=29 dirty=29 N2=29 7fe0bdba6000 prefer:3 file=/SYSV00000000\040(deleted)
7fff58d91000 prefer:2 stack anon=6 dirty=6 N2=6
```

Regarding the preceding example:

- The pages of the program executable remained on node 0 where they were cached. Also, the per-node page counts show that task 4559 touched only 12 pages of the executable, compared to task 4552, which touched 15 pages.

- The shared memory segment (`file=/SYSV000...`) shows the prefer:3 memory policy in both tasks' displays, even though the policy was set by only one of the tasks. The pages of this segment show node 3 for task 4552, which populated the segment by touching all of the pages. However, task 4559 shows no pages for that segment because the task has not yet touched any pages in the segment. Remember, the `numa_maps` output shows only those pages for which the task has page table entries.
- All of the other private (`anonymous or anon`) pages of both tasks were allocated on node 2, based on the task memory policy of each task.

You can use the `numactl` command to create shared memory segments and to set the shared memory policy for the segments it creates or for pre-existing segments. Because a memory policy only applies to pages allocated after the policy is applied, you must specify the policy before any pages have been allocated for the segment. You can meet this requirement by using `numactl` to create the segment. For example, the following command uses or creates the shared memory segment specified by `<shm-id>`:

```
numactl --shmid<shm-id> [--offset=<offset>] --length=<length><mem-policy-options>
```

When creating a segment, the `length=` parameter is required with the numactl command. For an existing segment, the `--offset=` and `--length=` parameters apply the memory policy to a subset range of the segment's pages.

For more information about `numactl` memory policy options, see the "Memory policy modes" section of this document or refer to the `numactl(8)` manpage.

Memory policy for page cache pages

Linux aggressively caches pages accessed from external storage devices in memory. Generally, the pages remain in memory where they were originally allocated until:

(1) The kernel needs to reuse the memory for other purposes

(2) When the file is truncated or deleted

(3) The file system is unmounted. (Cached pages are also freed if the `vm.drop_caches` kernel parameter is set to a value of 1 or 3, but this is uncommon in normal operation.) The collection of cached pages and the kernel subsystem that manages them is referred to as the page cache.

The policy that governs the allocation of page cache pages is always the task memory policy of the task on whose behalf the kernel allocated the page.

You can specify a VMA memory policy for a range of virtual address space into which a file has been mapped into memory (using `mmap`). However, that policy only applies to private, anonymous pages allocated when a task writes to a file-backed page cache page mapped privately to the task—that is, mapped without the `mmap` system call's `MAP_SHARED` flag. In this case, a private anonymous copy of the page is allocated according to any applied VMA memory policy. For shared page cache pages, only the task policy is applied and any VMA memory policy is ignored.

Most tasks run with the default memory policy. Therefore, page cache pages tend to be allocated on the node where the task executes. Local allocation is usually beneficial because the page cache page is usually copied immediately to a user buffer—most likely on the node where the task executes—or directly accessed through `mmap`.

However, consider a backup application that reads most or all of a large file system. In this case, local allocation for page cache pages can consume all of a node's memory, potentially forcing subsequent local allocations off-node. In this case, you can use `memory_spread_page` control to enable interleaving of page cache pages over several nodes, as explained in the "Using `cpusets` to control page cache and kernel memory allocator behavior" section.

Memory policy modes

A Linux NUMA memory allocation policy can be thought of as a tuple (an ordered list of elements) consisting of a memory policy mode that specifies the type of policy and an optional set of nodes from which the policy should attempt to allocate pages. Generally, these memory policy modes are most effective when used for designing or enhancing an application to adapt to a NUMA server's topology. However, some of the modes can be useful when you use numact1 to apply them to an application or shared memory area. These modes will be pointed out, where applicable, in the sections that follow.

Preferred mode

The preferred mode specifies a single node from which memory allocation should be attempted. However, if no free memory is currently available on the specified node, the preferred mode is allowed to overflow onto other nodes with available pages. For information about what happens when a page allocation request overflows to other nodes' memory, see the "Avoiding node memory overflow resulting in off-node allocations" section of this document.

The local allocation policy is a variant of the preferred memory policy, where the preferred node is the node of the CPU where the allocation request occurs. In fact, the way you specify the local allocation policy for the memory policy system calls is by specifying the preferred mode with an empty set of nodes.

The preferred mode is most useful for specifying the desired locality of a shared memory area or any other region of a task's virtual address space different from the task's local node—for example, to specify an I/O buffer on a remote node where an I/O bus is attached. Because you probably want the task executing on CPUs local to the preferred node, preferred mode is less useful as a task memory policy. Given this preference, simply affinitizing the task to the node's CPUs and using the local allocation policy would have the same effect while being simpler and less error-prone—involving a single constraint as opposed to two interdependent ones.

You can specify the preferred mode using the `numactl` command, as follows:

```
numactl --preferred=<nodeid> ...
```

Bind mode

The bind mode specifies a set of one or more nodes from which memory can be allocated. The way in which the kernel uses the set of nodes depends on the version of the kernel.

Older versions of Linux, such as found in Red Hat Enterprise Linux 5, process the set of nodes in order of their node ID numbers. This means that a bind mode allocation first attempts allocations from the lowest numbered node in the set. If that fails, it moves on to the next higher node ID. This algorithm completely ignores any internode distance information provided by the SLIT.

Newer versions of Linux, such as found in Red Hat Enterprise Linux 6 and 7, traverse the set of nodes based on the internode distances specified in the SLIT. This algorithm attempts to allocate memory on the node closest to the node from where the allocation is requested, within the constraints of the policy's set of nodes. If no SLIT is provided by the server, the behavior reverts to the numeric node ID algorithm.

Unlike preferred mode, bind mode does not allow overflow to nodes outside the set specified for the policy. Rather than overflow to another node, the kernel attempts to free up memory on the allowed nodes, possibly paging/swapping out some of the requesting task's other pages. For the application running under bind mode, the reclaim overhead and paging activity usually slow the program more than off-node references would. Thus, this mode is primarily useful only when you want to isolate an application or set of tasks so that it cannot affect tasks running elsewhere on the system, regardless of the effect on the bound tasks.

You can specify bind mode by using the `numactl` command, as follows:

```
numactl --membind=<node-list> ...
```

For information about another Linux mechanism for soft partitioning a large NUMA server and isolating applications from each other, refer to the "Using cpusets (privileged users only) to group tasks and constrain CPU/memory usage."

Interleave mode

The interleave mode specifies a set of nodes over which page allocations for a task or region of memory are spread. Why would this be useful, especially on a server where node interleaving in the system firmware is usually not advisable?

Interleave mode has two primary uses:

- Interleaving pages of a task or memory region over multiple nodes can improve performance through bandwidth sharing when:

  (1) Memory is accessed from CPUs on multiple nodes

  (2) No single best locality exists for the entire task or region. Examples of circumstances under which this approach would benefit performance include:

  – A multi-threaded task with more threads than fit on one node, or whose threads are distributed over multiple nodes for other reasons.

  - A shared memory area accessed by processes running on different nodes. However, local allocation policy might be a viable alternative. Tasks or threads on different nodes allocate their pages on first access locally under the default local allocation policy. Depending upon application behavior, the resulting distribution of shared memory pages might be adequate. Only experimentation reveals which allocation method is more beneficial for a given workload.
- For applications with memory regions that exceed the physical memory resources of a node—such as a large database's shared memory region—interleaving the region across several nodes can improve performance by preventing complete consumption of memory on one or more nodes for a single large region. (In this case, the other policies could result in complete consumption of the memory.) Interleaving leaves some local memory on each node—memory which might be needed for task private data and associated in-kernel data structures being accessed more frequently than shared memory regions.

Note
To avoid complete consumption of a single node's memory, use Linux interleave mode to balance the memory usage of large tasks or memory segments over multiple nodes.

When node memory overflow occurs, interleave mode behaves like the preferred mode. That is, interleave mode selects a node from which to allocate a page, and if that node has no available memory, the allocation is allowed to overflow to another node.

Specify interleave mode using the `numactl` command, as follows:

```
numactl --interleave=<node-list> ...
```

Note
Using interleave mode as the task memory policy of an application—for example, to balance the usage of a large memory segment created by the application—might not benefit the performance of the application itself. This is because the private pages of the application (such as its stack and heap) are interleaved as well. Thus, it is preferable for an application to interleave large segments explicitly, using VMA or shared memory policy.

Some large enterprise applications manage their memory locality transparently or provide configuration options to directly or indirectly specify the memory policy for large memory areas. Other applications might allow you to create shared memory areas explicitly, using `numactl--interleave=` and then specifying that the application use that area. For applications that provide neither of these controls, specifying an interleave memory policy for the task to balance large data areas can still improve overall system performance by allowing other smaller components of the application to perform better.

Avoiding node memory overflow resulting in off-node allocations

As discussed previously, all Linux memory policy modes except the bind mode allow the allocation to overflow or fall back to a different node when the node selected by the policy cannot satisfy the request. The kernel selects the fallback node from the selected node's list of nodes designated for overflow. These lists are called **zone lists**. The per-node zone lists are built by the kernel at boot time and are rebuilt any time the node/memory topology of the server changes.

When the server provides a populated SLIT, the kernel uses this information to order the zone lists such that each node overflows to its nearest neighbor. In the absence of a SLIT, the kernel substitutes a fake one that treats all nodes as equidistant from each other. In this case, the zone lists are ordered by node ID, so nodes do not necessarily overflow to a nearby node. If you have a server with a missing SLIT, the effects will only be seen on node overflow, and often the same NUMA tuning actions that you take to balance the load and improve locality also help to avoid node memory overflow.

Linux provides a couple of tunable kernel parameters that you can modify with the `sysctl` command to manage this overflow behavior. The following subsections describe these parameters, their defaults, and when you might want to use a non-default value.

Using the `vm.zone_reclaim_mode` kernel parameter to control node overflow

The `vm.zone_reclaim_mode` kernel parameter controls allocation behavior when a memory allocation request is about to overflow to another node. When `vm.zone_reclaim_mode` is disabled or zero, the kernel simply overflows to the next node in the target node's zone list.

When `vm.zone_reclaim_mode` is enabled, the kernel attempts to free up or reclaim pages from the target node's memory before going off-node for the allocation. For example, these pages might be cached file pages or other applications' pages that have not been referenced for a relatively long time. The allocation overflows only if the attempt to reclaim local pages fails.

The overhead of attempting to reclaim memory in the allocation path slows down that allocation. The kernel attempts to reclaim multiple pages at once, so the cost of the reclaim is amortized over several allocations. Nonetheless, it can still cause noticeable degradation for some workloads—especially for short-running jobs that cannot amortize the cost of the reclaim.

So why does Linux provide this option? What is the benefit?
For some long-running applications—for example, high-performance technical computing applications—the overall runtime can vary dramatically, based on the locality of their memory references. When such an application is started, even with well-thought out memory policies, a given node's memory could be filled up with page cache pages from previous jobs or previous phases of the application. Enabling `vm.zone_reclaim_mode` allows the application to reclaim those cached file pages for its own use, rather than going off-node. This action most likely benefits the application's performance over its remaining lifetime.

The default setting for `vm.zone_reclaim_mode` is enabled if any of the distances in the SLIT are greater than a fixed threshold, and disabled otherwise. Currently, the threshold in most shipping Linux distributions is a SLIT value of 20; this is the case for both Red Hat Enterprise Linux 5, 6, and 7. (The upstream kernel now uses a value of 30 as the threshold; this will appear in newer distribution versions.) For a server that does not supply a populated SLIT, `vm.zone_reclaim_mode` defaults to disabled because the remote distances in the kernel's default SLIT are all `20`. If the default setting is not appropriate for your workload, you can change it with the following command:

```
sysctl -w vm.zone_reclaim_mode={0|1}
```

Using the `vm.numa_zonelist_order` kernel parameter to control zone allocation on node 0
The `vm.numa_zonelist_order` kernel parameter controls the behavior when a page allocation request is about to overflow from the target node. This parameter is available in recent distributions such as Red Hat Enterprise Linux 6 and 7 but not in older distributions such as Red Hat Enterprise Linux 5. To help you understand the `vm.numa_zonelist_order` kernel parameter, this section examines the Linux memory zones in more detail. The Linux memory zones predate NUMA support. They arise from the x86 architecture. To support older I/O adapters that can only address 16 megabytes or 4 gigabytes of memory on servers that do not support I/O memory management units, Linux divides memory into zones. On the x86-64 architecture, Linux supports three zones:

- DMA zone—Contains the first 16 MB of physical memory. Drivers for adapters that can address only 16 MB of memory request from the DMA zone. No other memory suffices.
- DMA32 zone—Contains the first 4 GB of memory. Drivers for devices that are limited to 32-bit addressing request DMA32 memory. Again, they cannot use any higher-addressed memory, but they can use memory from the DMA zone if any is available there.
- Normal zone—Contains all physical memory above 4 GB. Most other kernel and user space allocations request memory from the normal zone. However, the requestors of these allocations work fine with memory from either of the other two zones.

Why are these zones of interest in a NUMA discussion? As indicated earlier, when a request for memory from one of the higher-addressed zones cannot be met, the request can be satisfied from a lower-addressed zone.

Consider an HPE ProLiant DL580 Gen10 Server with 8 GB of memory per node. Node zero contains all three zones:

- 16 MB of DMA zone
- 4 GB less 16 MB of DMA32 zone
- 4 GB of normal zone (the remaining memory)

The other nodes (1–3) contain all normal memory. When an application fills up the normal zone on node 0, to what node or zone should it fall back? If it falls back to the DMA32 and then DMA zones on node 0, all the available DMA memory could be used up. This can result in subsequent I/O requests failing because they cannot allocate the appropriate type of memory. On the other hand, if these allocations go off-node, the result could be much unused local memory on node 0 and longer average access latencies for the remotely allocated memory.

Linux resolves this conflict with the aid of the `vm.numa_zonelist_order` kernel parameter. If this parameter is set to node order, allocations fall back to lower-addressed zones in the same node. A zone order setting causes fallback to the same zone (for example, normal) on the next node in the zone lists. Default order uses the Linux default, which is:

- Node order if the DMA zones exceed one-half of the total memory or 70 percent of the node's memory
- Otherwise, zone order

You can override the `vm.numa_zonelist_order` kernel parameter using the following command:

```
sysctl -w vm.numa_zonelist_order={default|zone|node}
```

When this parameter is changed, the zone lists are rebuilt in the specified order.

You can also set this kernel parameter at boot time by adding the following to the bootloader's kernel command line:

```
numa_zonelist_order={default|zone|node}
```

For most x86-64 servers with 8 GB or more of memory per node running Red Hat Enterprise Linux 6, 7, or other recent Linux kernels, `numa_zonelist_order` defaults to zone order. If, as a result, you feel that your application is overflowing node 0 prematurely, and you are sure that your application will not interfere with I/O requests, you can change the zone list order to node order to keep more of your application's memory on node 0. Again, the other nodes have normal memory zones and they should not be affected by this parameter.

The default zone list ordering in Red Hat Enterprise Linux 5 resembles node ordering in Red Hat Enterprise Linux 6 and 7. Memory allocations on node 0 consume the normal and DMA32 zones before going to the next node in the zone list. Users with long-lived applications that fit comfortably in node 0 memory of a server that runs Red Hat Enterprise Linux 5 sometimes discover a performance drop. Especially, when the same server runs Red Hat Enterprise Linux 6 and 7 and the allocations spill over to another node. Setting `vm.node_zonelist_order` to node is a quick way to address this problem. A better long-term solution on Red Hat Enterprise Linux 6 might be to affinitize the application's memory usage with the techniques described in this document.

Controlling page migration

As discussed in the "Red Hat Enterprise Linux memory policy principle" section of this document, the kernel applies memory allocation policy when a page is allocated. In general, changes in memory policy do not affect the location of pages already resident in memory. However, a couple of exceptions to this rule exist:

- If the pages are not locked into memory (for example, with `mlock` or `mlockall`), the kernel might reclaim them, writing any dirty or anonymous pages to backing store. When the application next accesses a virtual address backed by one of these nonresident pages, the kernel reads the page contents from backing store into a newly allocated page that obeys the memory policy in effect for that virtual memory address. This is a form of migration by way of backing store.
- The kernel supports a number of system calls that allow a task to migrate pages mapped into its own address space or pages mapped in other tasks' address spaces between specified nodes.
  - The `mbind` system call, which installs a VMA memory policy, also accepts flags requesting that any pages in the specified range be migrated to obey the specified memory policy. A non-privileged task can migrate only those pages referenced by a single page table entry—that is, pages mapped only into this task's address space. A privileged (root) task can specify an additional flag to request that pages in the specified range be migrated, regardless of the number of tasks mapping the page. The `mbind` system call can migrate only those pages mapped into the calling task's address space. For more information about `mbind`, refer to the `mbind(2)` manpage.
  - The `migrate_pages` system call and its `numa_migrate_pages` library wrapper function allow a task to move to another specified set of nodes **all** of its own pages or **all** of another task's pages currently residing on one specified set of nodes. A command line version of this function is discussed later in this section.

For more information about `migrate_pages` and `numa_migrate_pages`, see the `migrate_pages(2)` and `numa_migrate_pages(3)` manpages.

For information on how Linux uses the equivalent of an internal version of `migrate_pages` to migrate the memory of tasks moved between cpusets or when resources of a `cpuset` are changed, refer to the "Using cpusets (privileged users only) to group tasks and constrain CPU/memory usage."

The `move_pages` system call and its `numa_move_pages` library wrapper function allow a task to move pages backing a specified list of virtual addresses to a set of nodes specified in a parallel list. The list of addresses can refer to the calling task's address space or to another task's address space. Obviously, this requires fairly intimate knowledge of the target task's address space layout. No command line interface currently exists for `move_pages`. For more information about `move_pages` and `numa_move_pages`, refer to the `move_pages(2)` and `numa_move_pages(3)` manpages.

Both `migrate_pages` and `move_pages` require the calling task to have the same privileges or relationship to the target task as are required to signal a task with the kill system call.

The kernel makes a best effort to migrate pages that satisfy the specified conditions. The kernel's attempt to migrate some or all pages might fail under certain conditions, such as insufficient resources on the target nodes, too many references to a page, and so forth.

Using the migratepages command to move a task's pages from one set of nodes to another

The `numactl` package that ships with most Linux distributions, including Red Hat Enterprise Linux 5, 6, and 7, installs the `migratepages` command. This command allows you to invoke the `migrate_pages` system call to move the pages of a specified task that reside on one set of nodes to another set of nodes. Use the command in the following format:

```
migratepages <pid> <from-node-list> <to-node-list>
```

This command moves the pages of the task specified by `<pid>` that reside on nodes specified by the `<from-node-list>` to nodes specified by the `<to-node-list>`. On an HPE ProLiant DL580 Gen10, the following three invocations are equivalent:

```
migratepages <pid> 0,2 1,3
migratepages <pid> !1,3 1,3
migratepages <pid> all 1,3
```

These invocations move all the pages from other nodes to node 1 and 3. The third invocation is equivalent to the others because Linux does not migrate pages that already reside on one of the target nodes.

When a non-privileged user invokes `migratepages`, the command migrates only those pages mapped solely by the specified task. When a privileged (root) user invokes `migratepages`, it attempts to migrate all pages mapped by the specified task that are on the `<from-node-list>` to the target node list; this includes all shared task and library executable segments and any shared memory segments. For more information about the `migratepages` command, refer to the `migratepages(8)` manpage.

Important
Take caution when migrating tasks that attach to large shared memory areas. These tasks can fill up the target nodes, potentially evicting pages of other tasks running on the target nodes.

Examples of the `migratepages` command

The following is a sample output excerpt of `numa_maps` for a test program started with the `numactl --cpunodebind=1` command. This excerpt forms the basis for the `migratepages` command examples to follow:

```
00400000 default file=/usr/local/bin/memtoy mapped=15 active=1 N1=15 00611000 default
file=/usr/local/bin/memtoy anon=1 dirty=1 N1=1 0068d000 default heap anon=28 dirty=28 N1=28
382d200000 default file=/lib64/libc-2.12.so mapped=150 mapmax=118 N0=150 382d397000 default
file=/lib64/libc-2.12.so
382d597000 default file=/lib64/libc-2.12.so anon=2 dirty=2 mapped=4 mapmax=49 N0=2 N1=2
382d59b000 default file=/lib64/libc-2.12.so anon=1 dirty=1 N1=1 382d59c000 default anon=3 dirty=3 N1=3
7f607f0d5000 default anon=262144 dirty=262144 N1=262144 7fffbb53a000 default stack anon=6 dirty=6 N1=6
```

All private (anon) pages are on node 1, as dictated by the default local allocation policy. Some of the `libc` shared pages reside on node 0 and node 1. The test program mapped a 1 GB (256K 4 KB pages) anonymous segment at `0x7f607f0d5000`; all of the segment's pages are on node 1.

Example: Non-privileged user

Assume a non-privileged user issues the following command, which attempts to migrate all of the pages of the previously mentioned task from node 1 to node 3:

```
migratepages <pid> 1 3
```

The test program's `numa_maps` output appears as follows:

```
00400000 default file=/usr/local/bin/memtoy mapped=15 active=1 N3=15 00611000 default
file=/usr/local/bin/memtoy anon=1 dirty=1 N3=1 0068d000 default heap anon=28 dirty=28 N3=28
382d200000 default file=/lib64/libc-2.12.so mapped=150 mapmax=121 N0=150 382d397000 default
file=/lib64/libc-2.12.so
382d597000 default file=/lib64/libc-2.12.so anon=2 dirty=2 mapped=4 mapmax=50 N0=2 N3=2
382d59b000 default file=/lib64/libc-2.12.so anon=1 dirty=1 N3=1 382d59c000 default anon=3 dirty=3 N3=3
7f607f0d5000 default anon=262144 dirty=262144 N3=262144 7fffbb53a000 default stack anon=6 dirty=6 N3=6
```

All the task's private pages have moved from node 1 to node 3. The 15 task executable pages mapped by the VMA at `0x00400000` have also moved from node 1 to node 3, because this is the only task that had these pages mapped. The evidence for this action is that the `numa_maps` output has no indication of a `mapmax=` item for this VMA, indicating that all of the pages shown are mapped only by this task. Non-privileged users are allowed to migrate shared pages that have a single page table reference.

Similarly, the two pages of `libc` on node 1 migrated to node 3. The `libc` pages on node 0 did not move because there was no request to migrate pages on node 0. Many of the pages are referenced by multiple tasks' page tables, as signified by the large value of `mapmax`. Therefore, the non-privileged `migratepages` would not have migrated those pages anyway.

Example: Privileged user

Let's assume the following command is invoked by a privileged user:

```
migratepages <pid> 0,3 2
```

This command specifies that task pages on nodes 0 and 3 be migrated to node 2. This command should move all of the `libc` pages on those nodes along with all private pages. The resulting `numa_maps` output includes the following lines:

```
00400000 default file=/usr/local/bin/memtoy mapped=15 active=1 N2=15 00611000 default
file=/usr/local/bin/memtoy anon=1 dirty=1 N2=1 0068d000 default heap anon=28 dirty=28 N2=28
382d200000 default file=/lib64/libc-2.12.so mapped=150 mapmax=121 N2=150 382d397000 default
file=/lib64/libc-2.12.so
382d597000 default file=/lib64/libc-2.12.so anon=2 dirty=2 mapped=4 mapmax=50 N2=4
382d59b000 default file=/lib64/libc-2.12.so anon=1 dirty=1 N2=1 382d59c000 default anon=3 dirty=3 N2=3
7f607f0d5000 default anon=262144 dirty=262144 N2=262144 7fffbb53a000 default stack anon=6 dirty=6 N2=6
```

Indeed, all the pages mapped by the task, including shared pages mapped by more than one task, are migrated to node 2.

Automating non-uniform memory access performance monitoring with numad

Red Hat Enterprise Linux 6 and 7 also include the `numad daemon`, which can be used as either a service or an executable to monitor the system NUMA performance characteristics. The daemon will attempt to detect long-running tasks and then move both their CPU and memory usage to the smallest set of NUMA nodes that can provide sufficient resources.

Tasks that complete quickly do not benefit from `numad`, so workloads consisting solely of such tasks should avoid the extra monitoring overhead by not enabling this service. The numad service uses all the techniques and methods previously detailed (such as `migratepages`, `numactl`, and others) to distribute workloads. For more information, refer to the Red Hat Enterprise Linux 7 Performance Tuning Guide.

Using `cpusets` to group tasks and constraint CPU/memory usage

Most of the Linux NUMA commands and programming interfaces for controlling memory allocation policy and page migration can be invoked by any user to execute tasks on, and allocate memory from, any node in the system, subject to resource availability. A Linux cpuset is a task-grouping mechanism. This mechanism allows a privileged administrator to constrain the NUMA resources (CPUs and memory) that tasks assigned to the cpuset can request. Cpusets are essentially a NUMA-aware version of what other operating systems (such as HP-UX) refer to as "processor sets."

The original interface to cpusets, as found in the Red Hat Enterprise Linux 5 and other older Linux distributions, was the `cpuset` pseudo-file system or `cpusetfs`. This file system is still available in Red Hat Enterprise Linux 6 and other recent Linux distributions, although newer interfaces have become available. To use this interface, you mount the cpuset file system on a mount point you select; `/dev/cpuset` is commonly used. (You might need to create this directory if your distribution does not provide it by default). The root of this file system is called the top-level cpuset, which includes all of the resources—CPUs and memory—in the system. The Linux init task starts out in the top-level cpuset; all other tasks inherit that cpuset unless changed by a privileged user.

You create additional cpusets to partition the system's resources by creating a directory in the `cpuset` file system. Each directory or cpuset in the `cpusetfs` contains a number of pseudo-files used to assign tasks to the cpuset and to specify the behavior of, and the resources available to, tasks assigned to the cpuset. As with the conventional Linux file systems, the cpuset file system supports a hierarchy of subdirectories. Thus, a cpuset can be subdivided into smaller subsets of system resources, or equivalent subsets with different behaviors. For more information about these controls, refer to the `cpuset(7)` manpage.

The cpuset task-grouping mechanism has been subsumed by Linux control groups. The cpuset is a client subsystem of control groups, accessed through the `cgroupfs` pseudo file system. The control groups file system interface is available in Red Hat Enterprise Linux 6 and other recent Linux distributions.

Other control groups client subsystems (known as controllers) support management of other resources such as memory (independent of locality), CPU cycles, swap space, and independent name spaces for system objects such as pids and System V IPC objects. These client subsystems, along with the task-grouping capability, form the basis for soft partitioning a single Linux instance into isolated containers. Further discussion of this aspect of control groups is outside the scope of this paper.

Control groups use the same directory hierarchy as do cpusets to name groups and to assign tasks to the groups. Within each directory, the cpuset-specific control files are prefixed with "`cpuset`." Other than this prefix, the usage of the cpuset control group and the legacy `cpusetfs` interface is the same. However, remember that a `cpusetfs` file system and a cgroupfscpuset controller cannot be mounted at the same time; you must pick one interface to use.

The examples that follow in this section use basic shell commands to create, populate, and manage cpusets. (Alternatively, Red Hat Enterprise Linux 6 and 7 provides a richer and more complex interface in the form of the user space commands delivered in the `libcgroup` package. For more information, refer to the "Red Hat Enterprise Linux 7 Resource Management Guide").

Assigning tasks to a cpuset is a privileged operation. Assigning resources to a cpuset and modifying the behavior of a cpuset also requires root privilege. Once a cpuset has been created and provisioned with a set of resources, tasks running in that cpuset are constrained by the cpuset resources. This means that any set of CPUs or nodes specified by `taskset` or `numactl`, or by one of the Linux scheduling APIs or NUMA APIs, is limited to the resources available to the cpuset. If the intersection of the set of resources requested and the resources available to the cpuset is empty, the command or call fails. Otherwise, the task uses the resources specified by the non-empty intersection. This means that the kernel does not allow you to install a memory policy that is invalid in the cpuset where the call is made. However, for an exception, see the "Sharing memory between cpusets" section of this document.

Using cpusets to isolate applications and constrain resource usage to specified nodes

Cpusets were originally added to Linux to partition large NUMA systems into subsets to isolate different applications from one another. Tasks executing in a cpuset are not allowed to specify, in any memory policy, a node that is not assigned to the cpuset allowed memories (mems) control file. Furthermore, memory allocations cannot overflow to a node outside the cpuset, regardless of the memory policy. So if the tasks in the cpuset exhaust the memory of all nodes assigned there, the tasks attempt to reclaim or swap out pages from only those nodes. If sufficient memory cannot be reclaimed, the task is killed.

As an example, assume that you have mounted the cpusetfs file system on `/dev/cpuset`. Now suppose you want to constrain an application to the resources of nodes 2 and 3 of an HPE ProLiant DL580 Gen10 Server on which hyper-threading is disabled. As root, you can create the cpuset using the following command:

```
mkdir /dev/cpuset/Mycpuset
```

You can then assign resources to Mycpuset using these commands:

```
echo 20-23,30-33 >/dev/cpuset/Mycpuset/cpus
echo 2,3 >/dev/cpuset/Mycpuset/mems
```

You can examine the resources assigned to the cpuset by reading the contents of the cpus and mems pseudo-files:

```
cat /dev/cpuset/Mycpuset/cpus 20-23,30-33
cat /dev/cpuset/Mycpuset/mems 2-3
```

Now, you can launch an application to run in the new cpuset as follows, still as root:

```
# move shell '$$' to Mycpuset su<app-user><app-start-command>
# invoke application as <app-user>
echo $$ >/dev/cpuset/Mycpuset/tasks
```

Typically, you would add these commands to a script that launches the application. All tasks and threads started by the

`<app-start-command>` inherit the cpuset and their CPU affinity, and memory allocation policies are constrained to the resources of the cpuset.

You can list the tasks assigned to a cpuset by reading the tasks pseudo-file:

```
cat /dev/cpuset/Mycpuset/tasks
```

You can query a specific task's cpuset using:

```
cat /proc/<pid>/cpuset
```

Kernel daemon tasks and, indeed, any tasks running in the top-level cpuset, remain free to allocate memory and execute on the CPUs of any node in the system. So if the goal of partitioning the system using cpusets is to isolate applications in the cpusets from all other processes, then you must create another cpuset containing a disjoint subset of the system's nodes (including both CPUs and memory) and move the remaining tasks from the top-level cpuset into that additional cpuset.

Using `cpusets` to control page cache and kernel memory allocator behavior

Earlier in this document, the "Memory policy for page cache pages" section explained how page cache pages use the task policy of the task that causes the file page to be read into memory. These pages remain in memory until forced out by memory pressure or an explicit request to free the kernel's caches. The same is true for kernel data structures. Generally, these allocations obey the task policy of the task for which the allocation was made. For example, the inodes that represent opened files remain cached in the inode cache until reclaimed or explicitly freed. Furthermore, the kernel memory allocator—any of various implementations of the so-called slab allocator—caches freed data structures for future allocations. Both of these types of allocations can consume large amounts of a node's memory, causing unexpected off-node page allocations or swapping activity.

One way to address this issue is to force caches to be dropped before running an application. The following command accomplishes this:

```
sysctl -w vm.drop_caches={1|2|3}
```

Note
The `vm.drop_caches` kernel parameter is not specifically related to cpusets, nor is it constrained by the cpuset from which it is invoked.

The effects of each of the three values you can specify for `vm.drop_caches` are as follows:

- Specifying 1 frees unused pages in the page cache.
- Specifying 2 frees up unused data structures in the slab caches, freeing any pages that become unused.
- Specifying 3 frees both.

Addressing the issue in this way is a very heavy-handed approach, as it drops caches system-wide—for example, requiring all applications to reread pages that they subsequently access. In addition, the effect of this approach is only temporary. Application startup could result in numerous files and data being read, thereby filling the caches again.

Cpusets provide two control files to spread out the page cache and kernel allocations over the nodes in a cpuset. This prevents applications that read a large amount of data, such as a backup program, from filling up one or more node's memories and forcing subsequent allocations off-node.

The `memory_spread_page` control file, when enabled, interleaves the page cache pages over the nodes in the cpuset, similar to the way that a task memory policy specifying interleave mode works. To enable this feature, you can use this command:

```
echo 1 >/dev/cpuset/Mycpuset/memory_spread_pages
```

Similarly, to spread/interleave kernel allocations over the nodes in a cpuset, use this command:

```
echo 1 >/dev/cpuset/Mycpuset/memory_spread_slab
```

You could enable page cache or slab interleaving for the entire system by setting these controls in the top-level cpuset at system initialization time. However, the default local page cache and kernel data structure allocations are optimal for many workloads—especially those involving many smaller, short-lived tasks. Note, however, that cpusets can overlap and share resources. You could create a cpuset that contains the CPUs and memory of all the nodes of the system, thus overlapping the top-level cpuset and enable page cache and/or slab interleaving for that cpuset. To spread out their memory load, you could run applications that are known to fill the page cache in that cpuset. Applications that benefit from local page cache allocations can remain in the top-level cpuset or in another cpuset that does not enable the interleaving.

Enabling page migration in cpusets

Earlier in this paper, the "Controlling page migration" section discussed system calls and commands that support migration of already-allocated pages to obey a newly installed VMA memory policy or to move pages without regard for memory policy. Cpusets provide an option to allow migration of a task's memory when the task is moved into a cpuset with a different set of allowed nodes, or when nodes are removed from a cpuset. In this case, the cpuset subsystem actually remaps the task's memory policies to make them valid in the new cpuset.

Enable cpuset migration using the following command:

```
echo 1 >/dev/cpuset/Mycpuset/memory_migrate
```

To enable migration when moving a task between cpusets, both the source and destination cpuset must have the `memory_migrate` control enabled.

Because modifying a cpuset's resources or moving tasks between resources is a privileged operation, any page migrations triggered by such a change are performed in the context of a privileged task. As a result, the kernel attempts to migrate all pages referenced by the affected tasks' page tables.

For example, suppose you want to move the Oracle log writer process out of a cpuset that contains two arbitrarily selected nodes (0, 1) of an HPE ProLiant DL580 Gen10 Server to a cpuset that contains only node 3. Because the storage is attached to node 3, you assume the process will perform better there. Furthermore, you have `memory_migrate` enabled in both cpusets because you want to migrate the log writer pages—stack, heap, and so forth—to keep them local to the task. As a result of this move, you will find that the kernel attempts to migrate all pages referenced by the log writer task's page tables, including pages in the Oracle executable image and pages of Oracle's typically large shared memory segments. Chances are good that node 3 will not have sufficient memory to hold all those pages. The kernel moves what it can to fill up node 3's memory, leaving the remaining pages behind.

Example: Modifying cpuset memory resources

Assume you create a cpuset named `Mycpuset` that includes the CPUs and memory from nodes 2 and 3 of an HPE ProLiant DL580 Gen10 Server. With this, you can enable the `memory_migrate control` for that cpuset and move your shell into Mycpuset and start the test program. The following is an excerpt from the test program's `numa_maps`:

```
00400000 default file=/usr/local/bin/memtoy mapped=15 active=1 N2=15 00611000 default
file=/usr/local/bin/memtoy anon=1 dirty=1 N2=1 01143000 default heap anon=29 dirty=29 N2=29
7f8ab2c29000 interleave:2-3 anon=261632 dirty=261632 N2=131072 N3=130560 7fff10b24000 default stack
anon=5 dirty=5 N2=5
```

Most of the test program's private pages, and even the shared executable pages, reside on Node 2—apparently where the program started. The test allocated a 1 GB anonymous segment mapped at `0x7f8ab2c29000`, and it interleaved the segment over both nodes in the cpuset. You can constrain the memory resources of the cpuset to just Node 3 using:

```
echo 3 >/dev/cpuset/Mycpuset/mems
```

After this change, examining the test program's numa_maps again would reveal the following:

```
00400000 default file=/usr/local/bin/memtoy mapped=15 active=1 N3=15
00611000 default file=/usr/local/bin/memtoy anon=1 dirty=1 N3=1
01143000 default heap anon=29 dirty=29 N3=29
7f8ab2c29000 interleave:3 anon=261632 dirty=261632 N3=261632
7fff10b24000 default stack anon=5 dirty=5 N3=5
```

All pages of the heap and stack and executable image that were on node 2 have been moved to node 3. Furthermore, the kernel reduced the set of nodes associated with the interleave policy of the 1 GB segment (VMA) at `0x7f8ab2c29000` to contain only the node that is valid in the modified cpuset and the pages of this segment that were on node 2 migrated to node 3.

You could have moved the cpuset's memory resources to a completely disjoint set, such as nodes 0 and 1. The kernel would then map the original set of nodes onto the new set of nodes and migrate the pages to the new set of nodes according to their original location. The next example illustrates this by moving the test program to a disjoint cpuset—a cpuset that shared no memory with the original one.

Example: Moving a task between cpusets

Assume the test program is in `Mycpuset` in the initial state described in the preceding example. Now you create a second cpuset named Mycpuset2 that contains the CPUs and memory resources from nodes 0 and 1. You also enable `memory_migrate` for `Mycpuset2`. Use the following command to move the test program from Mycpuset to Mycpuset2 by writing its pid to the target cpuset's tasks file:

```
echo<test-program's-pid>>/dev/cpuset/Mycpuset2/tasks
```

Examining the test program's `numa_maps`, you now see:

```
00400000 default file=/usr/local/bin/memtoy mapped=15 active=13 N0=15 00611000 default
file=/usr/local/bin/memtoy anon=1 dirty=1 N0=1 01143000 default heap anon=29 dirty=29 N0=29
7f8ab2c29000 interleave:0-1 anon=261632 dirty=261632 active=254989 N0=131072 N1=130560
7fff10b24000 default stack anon=5 dirty=5 N0=5
```

The pages of the program that were on node 2 moved to node 0, and those that were on node 3 moved to node 1.

Sharing memory between cpusets

Cpusets constrain the nodes from which tasks in the cpuset may allocate memory. They do not restrict access to any page that is already resident. For example, the kernel allows a client application task in one cpuset (cpuset-1) to access a database's shared memory region in another cpuset (cpuset-2) that shares no nodes or memory resources with the application's cpuset. How would this be beneficial?

As an example, you might want to run a web server in one cpuset and allow the web services to access a database server in another cpuset. Using cpusets, you could constrain the individual servers' memory to a sufficient subset of the system's nodes that are relatively close together to minimize NUMA effects. You could accomplish this using the taskset command or the `numactl --cpunodebind` command, as described in the "Constraining task migration to improve locality" section of this document. Cpusets would also ensure that if one of the servers tried to exceed the memory resources of its cpuset, the application would swap against itself without affecting the performance of other applications on the server. However, there are pitfalls that you must avoid to successfully share memory between cpusets.

As an example, assume that the database has been made NUMA-aware, and it applies a shared memory policy to interleave the shared memory region across the nodes in cpuset-2, perhaps to avoid overflowing any of the nodes' memory resources. If the database allocates all pages for the shared memory, either by initializing them or locking them into memory, the client tasks in cpuset-1 can access those pages (subject to permissions). On the other hand, if a page of the shared memory is first referenced by the client task, the kernel tries to allocate the page using the shared memory region's shared memory policy. The shared memory policy is used by all tasks that attach to the segment. The allocation would be forbidden because the shared policy contains no nodes that are valid in cpuset-1. Consequently, the kernel kills the client task because the page fault could not be satisfied with a successful page allocation (this is standard Linux behavior).

However, if instead, the shared memory region has a default memory policy, the client application can allocate the page and continue, but that page is then allocated on one of the nodes in the client's cpuset-1.

For access to shared memory regions from multiple cpusets, if you want any task to be able to allocate the page (for example, on the first touch), you must use a memory policy that is valid from all cpusets containing tasks that share the region. Otherwise, a task in the cpuset where the memory policy was installed must initialize/allocate all of the region's pages. This works because the kernel does not allow a memory policy that is invalid in the caller's context to be installed. Of course, the region must fit into its cpuset, and you must ensure it is not swapped out; otherwise, a client application in another cpuset could, again, touch a page that is not resident and try to allocate a page to swap it back in. In these situations, application writers should consider using `shmctl()` with the `SHM_LOCK flag or mlock()`.

Non-uniform memory access I/O locality

On an HPE ProLiant DL980 G7 Server, the NUMA locality of I/O adapters plays a big role in the I/O performance. Table 6 shows the locality of I/O adapters to the nodes.

Note

NUMA nodes 6 and 7 have no local I/O slots. As illustrated earlier in Figure 5, the QPI island of nodes 6 and 7 have no IOH—therefore, those nodes do not have I/O expansion connectivity.

Table 6. HPE ProLiant DL980 G7 I/O locality

Node	Slots 1–6	Slots 7–11	Slots 12–16
0, 1		X	
2, 3	X		
4, 5			X
6, 7			

The following commands show how to determine the NUMA locality of an I/O card such as an Ethernet adapter. Given the slot of an I/O card, the PCI bus address can be determined with the `dmidecode` command:

```
# dmidecode -t 9
Handle 0x090D, DMI type 9, 17 bytes System Slot Information
Designation: PCI-E Slot 14 Type: x4 PCI Express 2 x8 Current Usage: In Use Length: Short
ID: 14
Characteristics:
3.3 V is provided
PME signal is supported Bus Address: 0000:a7:00.0
For Ethernet adapters, an alternative is the ethtool command:
# ethtool -i eth12 driver: ixgbe version: 3.9.15-k
firmware-version: 0x80000306 bus-info: 0000:54:00.0 supports-statistics: yes supports-test: yes
supports-eeprom-access: yes supports-register-dump: yes supports-priv-flags: no
# find /sys -name numa_node | grep 54:00.0
/sys/devices/pci0000:50/0000:50:09.0/0000:54:00.0/numa_node
# cat /sys/devices/pci0000:50/0000:50:09.0/0000:54:00.0/numa_node
2
```

This command indicates that the Ethernet adapter is in one of Slots 1–6 with a NUMA locality of node 2. Slots 1–6 indicate node 2 as local, even though the IOH is connected to nodes 2 and 3. Similarly, Slots 7–11 indicate node 0 as local, even though the IOH is connected to nodes 0 and 1, and slots 12–16 indicate node 4 as local, even though the IOH is connected to nodes 4 and 5.

For best networking performance, all IRQs associated with a given I/O adapter must be distributed among the CPUs within a single NUMA node. By default, the `irqbalance` service is enabled and this service distributes IRQs among multiple nodes without paying attention to NUMA locality. For example, an Intel 10G adapter has 64 IRQs per port by default and these ports are distributed among CPU 0 to CPU 63 (nodes 0 to 6) on a 10-core eight Westmere CPU with HPE ProLiant DL980 G7 Server. It is best to disable irqbalance and then bind the IRQs to a single NUMA node.

For best storage performance, try to keep all IRQs from the I/O adapter bound to CPUs in the local NUMA node. In the earlier example, the following commands bind IRQs 170–173 of the adapter to CPU 20–23 on NUMA node 2:

```
#echo 100000 > /proc/irq/170/smp_affinity
#echo 200000 > /proc/irq/171/smp_affinity
#echo 400000 > /proc/irq/172/smp_affinity
#echo 800000 > /proc/irq/173/smp_affinity
```

To stop and disable the irqbalance daemon, issue the following commands in RHEL 6:

```
# service irqbalance stop
# chkconfig irqbalance off
```

Or the following commands in RHEL 7:

```
# systemctl stop irqbalance
# systemctl disable irqbalance
```

Non-uniform memory access considerations for kernel-based virtual machine guests

When guests are provisioned using `libvirt`, the hypervisor's default policy is to schedule and run the guest on any available resources on the host. As a result, the resources backing a given guest could end up getting spread out across multiple NUMA nodes. Over a period of time, the resources might get moved around, leading to poor and unpredictable performance inside the guest.

To avoid this problem, it is important to explicitly restrict the guest's vCPUs to run on a given set of physical CPUs (that is, ideally contain the guest to run within a given host NUMA node). In other scenarios, the compute requirements of a given workload might dictate the need for provisioning larger-sized guests (larger than a single-host NUMA node). It is important to ensure that such larger-sized guests are appropriately pinned across the desired host NUMA nodes. For such larger guests, it is also important to consider exposing virtual NUMA nodes within the guest so that the OS instance running within the guest can make NUMA-aware choices.

In Red Hat Enterprise Linux 6 and 7, there is support available in `libvirt` to accomplish all these goals, as illustrated in the following examples. In addition, enhancements were made to enable `numad` to better control the resource allocation of guests, which might help in certain use cases. For more details, refer to the documentation in the manpages for `virsh` and `numad`, as well as the "Red Hat Virtualization Tuning and Optimization Guide".

The first example is of a larger guest (20 vCPU/256 GB) hosted on an HPE ProLiant DL980 G7 platform that has hyper-threading turned off. The vCPUs in the guest's XML configuration file (this can be examined via `virsh`) look something like:

```
<vCPU placement='static'>20</vcpu>
```

These 20 vCPUs could be scheduled to run on any of the host's physical CPUs. If the user wants to restrict the guest VCPUs to run on only a specific set of physical CPUs, the user can do so by specifying the `cpuset` attribute in the guest's XML file. For example, the 20 vCPUs of the guest need to be run on the physical CPUs 0–19 (in other words, the host NUMA Nodes 0 and 1):

```
<vcpuplacement='static' cpuset='0-19'> 20 </vcpu>
```

This command will enable `libvirt` to set the affinity of the threads representing the 20 vCPUs to run on any of the physical CPUs 0–19. By using the cputune attribute, you can even choose to pin each individual vCPU to an individual physical CPU, as follows:

```
<vcpu placement='static'>20</vcpu>
<cputune>
<vcpupinvcpu='0' cpuset='0'/>
<vcpupinvcpu='1' cpuset='1'/>
<vcpupinvcpu='2' cpuset='2'/>
<vcpupinvcpu='3' cpuset='3'/>
<vcpupinvcpu='4' cpuset='4'/>
<vcpupinvcpu='5' cpuset='5'/>
<vcpupinvcpu='6' cpuset='6'/>
<vcpupinvcpu='7' cpuset='7'/>}
<vcpupinvcpu='8' cpuset='8'/>
<vcpupinvcpu='9' cpuset='9'/>
<vcpupinvcpu='10' cpuset='10'/>
<vcpupinvcpu='11' cpuset='11'/>
<vcpupinvcpu='12' cpuset='12'/>
<vcpupinvcpu='13' cpuset='13'/>
<vcpupinvcpu='14' cpuset='14'/>
<vcpupinvcpu='15' cpuset='15'/>
<vcpupinvcpu='16' cpuset='16'/>
<vcpupinvcpu='17' cpuset='17'/>
<vcpupinvcpu='18' cpuset='18'/>
<vcpupinvcpu='19' cpuset='19'/>
</cputune>
```

Starting with Red Hat Enterprise Linux 6, the kernel-based virtual machine (KVM) guests are by default backed by Transparent Huge Pages (THP). You can use the `numatune` attribute to specify the list of host NUMA nodes that contribute THPs for backing a guest. For uniform memory access latency, and for better utilization of the underlying system's memory bandwidth, it might be better to consider specifying "interleave" across the NUMA nodes. In the following example, the guest's backing memory will come from host NUMA Nodes 0 and 1:

```
<numatune>
<memory mode='interleave' nodeset='0-1'/>
</numatune>
```

Exposing virtual NUMA nodes in the guest can also be accomplished via `libvirt`. In the following example, two virtual NUMA nodes each are exposed with 10 CPUs and 128 GB of memory to the guest:

```
<cpu>
<numa>
<cell cpus='0-9' memory='134217728'/>
<cell cpus='10-19' memory='134217728'/>
</numa>
</cpu>
```

Automatic NUMA balancing (Red Hat Enterprise Linux 7)

Starting with Red Hat Enterprise Linux 7, a new kernel feature called Automatic NUMA balancing was introduced. This kernel feature is enabled by default and is activated when the kernel is booted on systems with multiple NUMA nodes. The goal of this feature is to help improve out-of-box performance of workloads running on NUMA systems.

When the feature is active, the kernel's scheduler will scan through each task's address space and revoke access permissions to chunks of pages in active use by the task. Page faults resulting from attempts to access these pages are trapped and resolved by the scheduler and various fault statistics (for example, local node memory fault vs. remote node memory fault, and whether the pages were being accessed by a single task [private fault] or a group of tasks [shared fault]) are gathered on a per-task and per-NUMA-node basis. The scheduler uses these statistics to reach decisions about migrating memory to the appropriate NUMA nodes. The scheduler also works with the load balancer to reach decisions on task placement/re-balancing across the NUMA nodes to ensure efficient access to memory. For cases where the workload is large enough to force the associated tasks and memory to span multiple NUMA nodes, the scheduler tracks the private and shared faults and migrates pages between the NUMA nodes as necessary to help maximize memory bandwidth utilization.

Automatic NUMA balancing helps improve performance for different workloads on a variety of NUMA systems. In most cases, this improved performance is very close to the performance that can be achieved via optimal manual binding. However, this feature is new and evolving, and it will mature over time.

Note

On systems with complex NUMA topologies, the scheduler algorithm must be further enhanced to account for the internode distances and choose the best nearby nodes for placing the larger workloads that span multiple nodes.

Use cases where workloads rely on memory that cannot be migrated (that is, `mlock'd` or backed by `hugetlbfs` and more) must be addressed by this feature. In addition, this feature currently focusses on CPU and memory localities; it does not yet consider the I/O interrupt locality.

Use cases are likely to exist where manual binding is still the preferred method for getting the best performance. Attempts to manually bind a workload (or using `numad` to do the same) will override the automatic NUMA balancing feature for that workload.

Users running on Red Hat Enterprise Linux 7 can verify if the feature is active on a given system using the `sysctl` command:

```
# sysctl kernel.numa_balancing kernel.numa_balancing=1
```

For a given use case, if you want to completely turn off the automatic NUMA balancing feature on the system, you can do so by using sysctl:

To disable:

```
# sysctl -w kernel.numa_balancing=0
```

To re-enable:

```
# sysctl -w kernel.numa_balancing=1
```

KVM is one of the primary beneficiaries of the new automatic NUMA balancing feature in Red Hat Enterprise Linux 7. Pinning the VMs backing resources (as discussed in the section on "NUMA considerations for KVM guests") yields better performance, but it creates issues while attempting to live-migrate the VM between hosts. The target host might not have the same set of backing resources available, or the target host might have a totally different NUMA topology.

Automatic NUMA balancing helps find the best backing resources for the VM on any NUMA host running Red Hat Enterprise Linux 7—thereby helping to improve the out-of-box performance of KVM guests of different sizes. However, in certain use cases where the VM is being backed by memory that cannot be moved, the automatic NUMA balancing feature cannot help (for example, VMs relying on `hugetlbfs` for backing memory instead of THPs; VMs that require all their memory to be pinned, use Virtual Function I/O [VFIO], or run applications that have real-time requirements).

If the VM is configured to have multiple virtual NUMA nodes and if the guest OS instance is Red Hat Enterprise Linux 7, then automatic NUMA balancing would be enabled within the guest OS, and workloads running in that guest instance would benefit from the feature. Depending on the specific use case, a user could also choose to explicitly bind their workload(s) to the virtual NUMA nodes in that guest OS instance and override the automatic NUMA balancing feature.

References

Manpages for commands:

`taskset(1) numactl(8) sysctl(8) migratepages(8)`

Manpages for libraries:

`set_mempolicy(2) get_mempolicy(2) mbind(2) sched_setaffinity(2) sched_getaffinity(2)`

Manpages containing general information:

`numa(7) cpuset(7) proc(5)`

HPE ProLiant servers: hpe.com/us/en/product-catalog/servers/proliant-servers.hits-12.html

HP ProLiant DL980 G7 server with HP PREMA Architecture white paper

Red Hat Enterprise Linux 7 performance tuning guide

Red Hat Enterprise Linux 7 virtualization tuning and optimization guide

Documentation feedback

Hewlett Packard Enterprise is committed to providing documentation that meets your needs. To help us improve our documentation, send any errors, suggestions, or comments to documentation feedback (docsfeedback@hpe.com). When submitting your feedback include the document title and part number, version number, or the URL.

Learn more at

hpe.com/info/servers

Sign up for updates

a00039147ENW, January 2018