Linux NUMA support for HP ProLiant servers

Technical white paper

Table of contents

Abstract

This white paper discusses Linux support for HP ProLiant servers with Non-Uniform Memory Access (NUMA) topologies as delivered in the Red Hat Enterprise Linux (RHEL) and other Linux distributions (distros). 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 understanding how Linux manages these resources on a NUMA system.

Intended audience

This white paper is intended for experienced HP and customer personnel who design application systems and analyze and optimize the performance of those applications on HP ProLiant NUMA servers running Linux. The information in this white paper applies both 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 servers

The Linux kernel gained support for cache-coherent Non-Uniform Memory Access (NUMA) systems in the Linux 2.6 release series. The kernel's support for NUMA systems 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 Linux distros such as RHEL5 and RHEL6 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 command line 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 the tools available to examine and affect this behavior. This section discusses the benefits of using NUMA servers and the implications their use has for both kernel and application software.

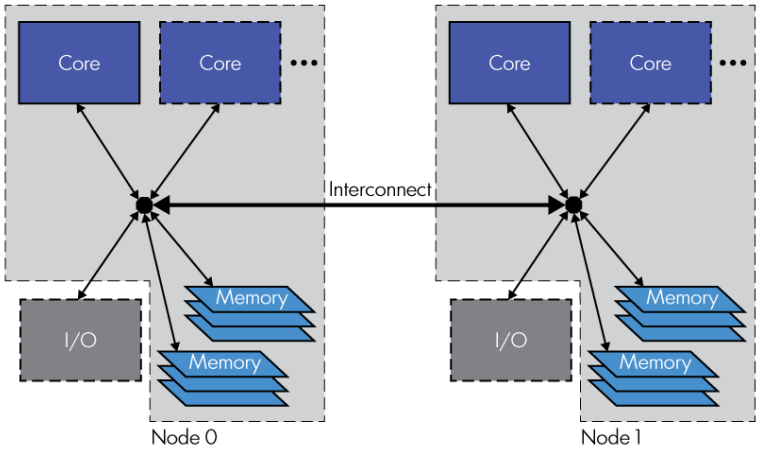
HP incorporates the NUMA architecture in its servers primarily because of the architecture’s capability for providing scalable memory and I/O bandwidth for today's high core count, high performance processors within the constraints of physical and economic feasibility. In fact, with memory controllers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

NUMA server topology

A NUMA server can be viewed as a collection of SMP/SMT building blocks, each containing a subset of the server's CPUs, memory, and I/O connections. These building blocks are often referred to as nodes or cells; in 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 intranode 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 intranode 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 internode 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 intranode 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 nonlocal reference — a reference to memory existing on a node remote to the reference-issuing node.)

Interconnects 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 of those streams 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 NUMA-aware software is to reduce the consumption of internode 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.

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

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

Linux kernel default NUMA behaviors

Ideally, system software developers modify the kernel, libraries, compiler toolchain, and other components to achieve optimal performance by default in a wide array of circumstances. Primarily this means providing reasonable (that is, nonpathological) 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 Linux kernel's NUMA features evolved throughout the 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 Linux servers configured for NUMA often surpassed that 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 Linux 3.X releases, default Linux NUMA behavior continues to improve and keeps pace with the newest NUMA server topologies.

However, because memory access patterns and the effects on application performance can be workload dependent, 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, 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 memory allocation policy, which may be configured uniquely for different tasks and even for different memory regions. (For more information, see the "Linux memory policy principles" section.) Linux also provides mechanisms

[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 all the system's NUMA characteristics. Users are often surprised by this. For more information, see the "Firmware-configured node interleaving" section.

that can constrain the set of hardware resources an application may 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 best and where detrimental performance interactions can be avoided. (For information about one of the ways to do this, see the "Using cpusets" section.) These mechanisms fall into two categories:

- Inheritable task attributes (across `fork()/exec()`) or attributes that can be applied to a running task. These can be used 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, of course, require source modifications, recompilation, and/or relinking.

HP NUMA servers

This section examines selected HP ProLiant servers that have a NUMA architecture. It provides the context for understanding Linux NUMA features and introduces some commands and concepts that are explained in more detail in subsequent sections.

Prevalence of the NUMA architecture

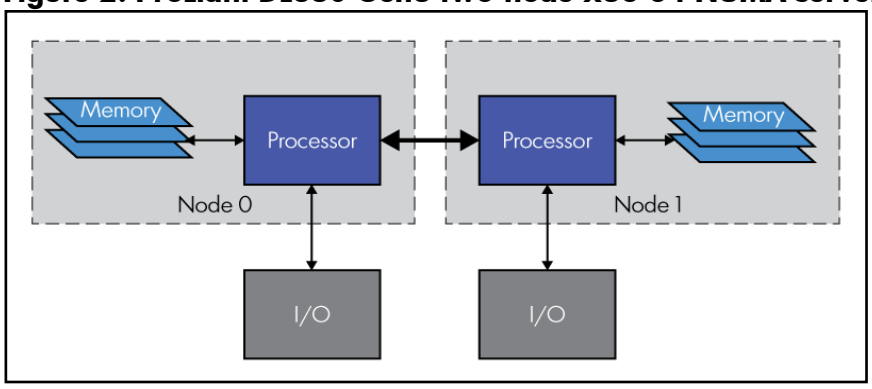
High-end enterprise and technical computing servers have long exhibited NUMA characteristics. High-end HP PA-RISC and Integrity servers use NUMA architecture to increase their memory scalability for support of large numbers of high performance processors. With the introduction a few years ago of AMD processors containing HyperTransport (HT) links, commodity x86-64 two- and four-node 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. HP ProLiant multi-socket servers based on Intel QPI or AMD HT processors are inherently NUMA systems.

This document uses the HP ProLiant DL380 Gen8, DL580 G7, and DL585 G6 servers as examples, as each of them exhibit unique NUMA topologies.

HP ProLiant DL380 Gen8 servers

Figure 2 shows a simple schematic diagram of a ProLiant DL380 Gen8 with its two Intel multi-core processors. Each processor with its associated memory is a Linux NUMA node, and each contains four, six, or eight processor cores, depending upon the model. The bold line connecting the nodes in the diagram represents a QuickPath Interconnect (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 QPI and exhibit higher latencies and proportionally lower bandwidth than do references to memory on the local node.

Figure 2: ProLiant DL380 Gen8 two-node x86-64 NUMA server



Obtaining information about DL380 Gen8 hardware resources and NUMA topology

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.

```
dl380> numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7
node 0 size: 8157 MB
node 0 free: 7326 MB
node 1 cpus: 8 9 10 11 12 13 14 15
node 1 size: 8191 MB
node 1 free: 7746 MB
node distances:
node   0   1
  0:  10  20
  1:  20  10
```

The nodes on this system contain eight processor cores each, which Linux identifies as CPUs. Note that the CPU ids used by Linux to represent the processor cores are assigned in sequence within each node, starting with node 0 and then within the next sequentially numbered node id. This is a characteristic of HP ProLiant Intel NUMA servers and differs from how CPU ids are assigned by ProLiant AMD servers, as will be discussed subsequently.

Processor hyperthreading is not enabled on this system. The following shows an `numactl` example where hyperthreading is enabled:

```
dl380> numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 16 17 18 19 20 21 22 23
node 0 size: 8157 MB
node 0 free: 6950 MB
node 1 cpus: 8 9 10 11 12 13 14 15 24 25 26 27 28 29 30 31
node 1 size: 8191 MB
node 1 free: 7086 MB
node distances:
node   0   1
  0:  10  20
```

```
  1:  20  10
```

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 or BIOS in a table called the **System Locality Information Table** or **SLIT**. More information about SLIT data is available in the "Avoiding node memory overflow resulting in off-node allocations" section.

Analyzing the DL380 Gen8 NUMA topology and internode bandwidth

Not all HP ProLiant x86-64 NUMA servers have a populated SLIT. The DL380 Gen8 is an example. When you see a node distance table containing all 10s and 20s, as in the preceding example, this usually means that the system's BIOS 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 both 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 the DL380 Gen8, as shown in the following internode bandwidth table.

Table 1 shows the internode bandwidth of this DL380 Gen8 with 2.40 GHz eight core Xeon E5-2600 processors for memory loads and stores (combined read/write workload), as performed with the `memcpy` function. Each row in the table shows the results for 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: DL380 Gen8 internode R/W bandwidth (memcpy)

Memory on: ↓Task on:	Node 0	Node 1
Node 0:	1.00	0.55
Node 1:	0.55	1.00

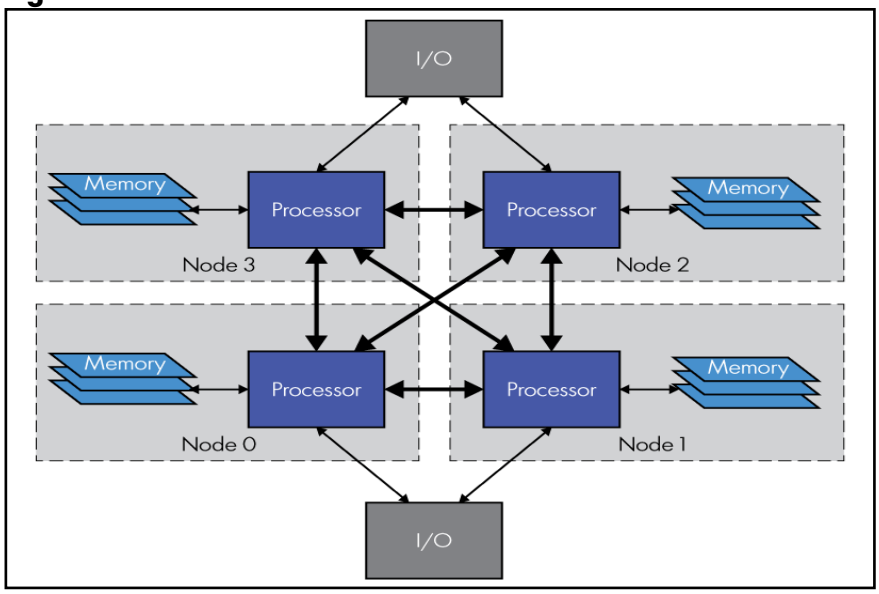
To show the relative cost of off-node references, the values in the table are normalized to the average local bandwidth of all the nodes.

The memory bandwidth measured for remote accesses was 55% 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, and 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 "Constraining task migration to improve locality" section that follows discusses how to bind tasks to a set of CPUs to constrain memory locality.

HP ProLiant DL580 G7 servers

Figure 3 shows a simple schematic diagram of a ProLiant DL580 G7 consisting of four Intel multi-core processors. Each processor and its associated memory is a Linux NUMA node, and each processor contains four, six, eight, or ten processor cores, depending upon the model. All of the nodes are connected by QPI links 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 3: ProLiant DL580 G7 four-node x86-64 NUMA server



Obtaining information about DL580 G7 hardware resources and NUMA topology

As in the case of the DL380 Gen8, the Linux kernel's view of its 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
node 0 size: 16373 MB
node 0 free: 15322 MB
node 1 cpus: 10 11 12 13 14 15 16 17 18 19
node 1 size: 16384 MB
node 1 free: 15905 MB
node 2 cpus: 20 21 22 23 24 25 26 27 28 29
node 2 size: 16384 MB
node 2 free: 15823 MB
node 3 cpus: 30 31 32 33 34 35 36 37 38 39
node 3 size: 16383 MB
node 3 free: 15941 MB
node distances:
node   0   1   2   3
  0:  10  20  20  20
  1:  20  10  20  20
  2:  20  20  10  20
  3:  20  20  20  10
```

This system has ten cores or CPUs per node and is not hyperthreaded. The following shows an example of output for a system with hyperthreading enabled:

```
dl580> 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: 16373 MB
node 0 free: 15631 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: 16384 MB
node 1 free: 15587 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: 16384 MB
node 2 free: 15916 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: 16383 MB
node 3 free: 15804 MB
node distances:
node   0   1   2   3
  0:  10  20  20  20
  1:  20  10  20  20
  2:  20  20  10  20
  3:  20  20  20  10
```

Analyzing the DL580 G7 NUMA topology and internode bandwidth

The DL580 G7 firmware does not provide a populated SLIT in its default configuration. As shown in the example, Linux uses its default node distance values of 10 and 20 to construct a SLIT that treats all remote nodes as equidistant from any other node. This accurately reflects the system's topology, given that all its nodes are fully connected. Furthermore, measurements of the system's behavior confirm this, as seen in Table 2.

Note

There is an RBSU configuration option on the DL580 G7 named "System Locality Information Table" which, when enabled, causes the firmware to supply a populated SLIT. (This option is also accessible through the system firmware's graphical console interface.) For system ROM versions earlier than P65 11/02/2011, the firmware provides a table containing remote node distance values of 21. For firmware revisions equal to or later than that, the table contains remote node distance values of 11. HP recommends that you avoid using the RBSU option when running Linux on the DL580 G7, even though the firmware's graphical interface prompt suggests the option has performance benefits. Using this option on the DL580 G7 can lead to unexpected system behavior. The originally-intended effect is better obtained by using a Linux kernel tuning knob discussed in the "Using the `vm.zone_reclaim_mode`" section of this document.

Table 2 shows the internode bandwidth of this DL580 G7 with 2.40 GHz ten core Xeon E7 4870 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: DL580 G7 internode R/W bandwidth (memcpy)

Memory on: ↓Task on:	Node 0	Node 1	Node 2	Node 3
Node 0:	1.0	0.70	0.68	0.68
Node 1:	0.70	1.01	0.70	0.69
Node 2:	0.69	0.70	1.01	0.69
Node 3:	0.69	0.70	0.69	0.99

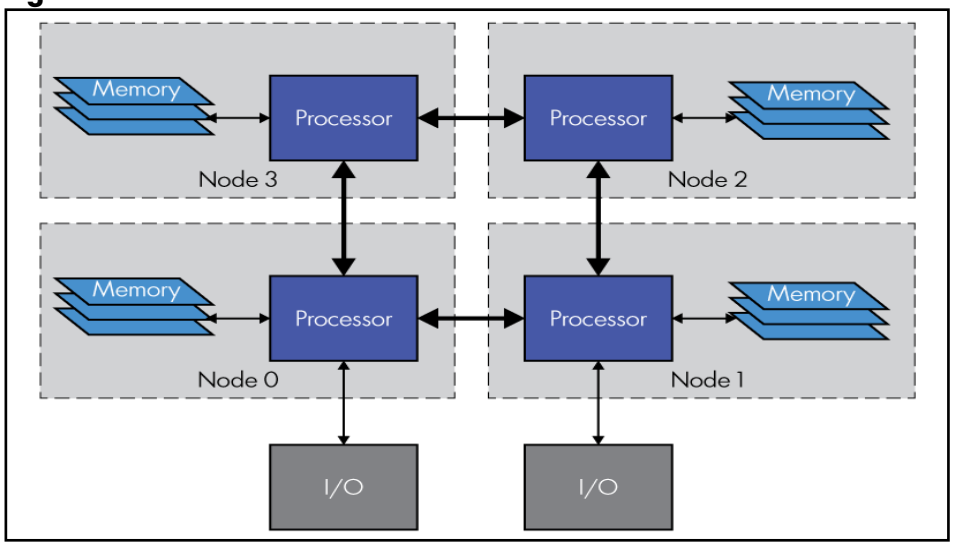
To show the relative cost of off-node references, the values in the table are normalized to the average local bandwidth of all the nodes. The table shows minor variations in these bandwidth measurements, depending on the node where the task accessing the memory is running. For any given test run, some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

The table shows clearly that the remote access bandwidth for all nodes is roughly 68 to 70% of the local bandwidth for mixed reads and writes. The uniformity of these results agrees well with the system's physical fully-connected topology. The Linux kernel's default SLIT is a fair approximation in that it correctly represents all remote nodes as a single hop away, but it slightly overstates the penalty for off-node allocations. The kernel's default SLIT is a suitable approximation for the DL580 G7 because the kernel's NUMA policy code is more sensitive to the gross structure of the topology (connectivity and hop count) than to the internode latency differences shown in the SLIT.

HP ProLiant DL585 G6 servers

Figure 4 shows a simple schematic diagram of a ProLiant DL585 G6, with its four AMD multi-core processors. This is an older server that is still in service with many HP customers. It presents a useful example because its topology differs slightly from the newer HP ProLiant servers. Each of its processors and the associated memory form a Linux NUMA node, and each processor contains six cores. Unlike the DL580 G7, the DL585 G6 does not have a fully-connected NUMA topology. Instead, its nodes are connected in a ring by HyperTransport (HT) links, 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 4: ProLiant DL585 G6 four-node x86-64 NUMA server



Obtaining information about DL585 G6 hardware resources and NUMA topology

The following `numactl --hardware` command output reflects the Linux kernel view of this server's topology:

```
dl585> numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 4 8 12 16 20
node 0 size: 24204 MB
node 0 free: 11233 MB
node 1 cpus: 1 5 9 13 17 21
node 1 size: 24240 MB
node 1 free: 14528 MB
node 2 cpus: 2 6 10 14 18 22
node 2 size: 24240 MB
```

```
node 2 free: 14503 MB
node 3 cpus: 3 7 11 15 19 23
node 3 size: 24240 MB
node 3 free: 14341 MB
node distances:
node    0   1   2   3
   0:  10  20  20  20
   1:  20  10  20  20
   2:  20  20  10  20
   3:  20  20  20  10
```

Note that the CPU ids that Linux uses to represent the processor cores on this server are interleaved over the nodes so that the lowest number CPU id on each node equals the node id. This numbering convention is characteristic of HP ProLiant AMD NUMA servers, including the most recent Gen8 systems.

Analyzing the DL585 G6 NUMA topology and internode bandwidth

The DL585 G6's firmware does not provide a populated SLIT. Instead, Linux uses the default distance values of 10 and 20 to construct a SLIT, treating all remote nodes as equidistant from any other node. Obviously, this does not accurately reflect the topology of the DL585 G6 described previously. However, it is a workable approximation, as confirmed by Table 3.

Table 3 shows the internode bandwidth of a DL585 G6 with 2.8 GHz six-core AMD 8439 SE 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: DL585 G6 internode R/W bandwidth (memcpy)

Memory on: ↓Task on:	Node 0	Node 1	Node 2	Node 3
Node 0:	1.0	0.8	0.8	0.6
Node 1:	0.9	1.1	0.6	0.9
Node 2:	0.9	0.6	1.0	0.9
Node 3:	0.7	0.8	0.8	1.1

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.

There are some variations in the ratios of local to average local and remote 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.

The local bandwidth measurements are on the left to right descending diagonal of Table 3 (shaded), and the two-hop remote bandwidth measurements are on the left to right ascending diagonal. The remaining cells of the table show one-hop remote bandwidth measurements. Memory bandwidth to nodes a single hop away is 80 to 90% of the local bandwidth, while bandwidth to nodes two hops away is 60 to 70% of the local bandwidth. Note that two-hop remote references on this topology incur latencies and consume bandwidth on two HyperTransport links, potentially increasing link contention across the topology. On a busy system, this can lead to additional performance impact.

You can improve performance significantly by using this topology information to tune the system. Such tuning techniques are discussed later in this document. The default SLIT that Linux constructs causes

the kernel to favor local memory allocations over allocations to remote nodes in all cases, but the kernel treats all remote nodes the same. This can lead to less than optimal placement for some allocations, depending upon the circumstances. In circumstances where there might be insufficient memory to satisfy allocation requests on the local node, performance can be improved by tuning a DL585 G6 to preferentially allocate memory on the adjacent nodes rather than on the node two hops away. This consideration is especially important when all the memory required by an application cannot fit on the local node.

Firmware-configured node interleaving on ProLiant servers

On many HP ProLiant servers, the ROM-Based Setup Utility (RBSU) allows enabling or disabling a configuration option called Node Interleaving. (The same 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 BIOS configures the hardware to interleave sequential physical page addresses over all the nodes/sockets in the system. Upon boot, the BIOS 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 Linux kernel, such as found in RHEL4, 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 RHEL5 and RHEL6, node interleaving is seldom advantageous, especially on larger system configurations. In some cases, it 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 these 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 HP recommends that you **do not** enable firmware node interleaving on ProLiant servers running RHEL5 or RHEL6. First try to optimize your system with the tuning methods discussed in the remainder of this document. These 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.

Linux memory policy principles

This section introduces the principles underlying the 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 maximize 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 both 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, a DL580 with 256 GB of memory has over 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-node DL580, Linux manages the pages in four pools of 16 million pages each. That may be a large number of pages to manage, but the disabled interleaving improves scalability: Linux's NUMA features result in a smaller set of CPUs—those local to the node—accessing the node's lock, and they access the local data structures it protects much more often than do remote CPUs.

Linux defaults to the local allocation policy

The mechanism that Linux uses to direct memory requests to the page allocator of the local node is called the **local allocation policy**. The local allocation policy 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 or user or by 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 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.

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.

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>
```

Or, 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 by 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 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 a DL580 G7 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 of these cases, the indicated tasks are bound to the specified set of CPUs. If these tasks are all using 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, best locality can be achieved by setting an application's CPU affinity at `fork` time, as with the first form of `taskset` or using `numactl --cpunodebind`. This guarantees that all kernel data structures associated with the applications' component tasks, and all of the tasks' private data pages, are located on or near the nodes containing the specified CPUs and 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" section.

How to use 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 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`:[2]

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

The shell parameter $$ evaluates to the shell's process id. For a `bash` shell on RHEL6, this displays over 30 lines of information. This discussion considers only those lines containing the text `bash`, `stack`, `heap`, and `libc`:

```
00400000 default file=/bin/bash mapped=138 active=114 N0=135 N2=3
006d3000 default file=/bin/bash anon=10 dirty=10 active=9 N0=2 N1=3 N2=1 N3=4
008dc000 default file=/bin/bash mapped=2 N0=2
00cd5000 default heap anon=63 dirty=63 active=61 N0=17 N1=18 N2=12 N3=16
382d200000 default file=/lib64/libc-2.12.so mapped=148 mapmax=139 N0=3 N2=145
382d397000 default file=/lib64/libc-2.12.so
382d597000 default file=/lib64/libc-2.12.so anon=1 dirty=1 mapped=4 mapmax=75
   N2=3 N3=1
382d59b000 default file=/lib64/libc-2.12.so anon=1 dirty=1 N0=1
7fff1c876000 default stack anon=8 dirty=8 N0=1 N1=2 N2=1 N3=4
```

[2] Although this command may be used safely in testing to investigate application behavior, HP and Red Hat do not recommend its use on production systems because of possible performance side effects. This is especially true if the command is used at high frequency, as from within a script. The kernel operations performed are CPU-intensive and will cause page table modifications to block while they are in progress, potentially affecting other processes running on the system.

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 /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 in a previous section, the system uses 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 that 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, and would 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 is not displayed unless its value differs from the anon= and dirty= page counts, discussed subsequently. Note that 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 be displayed 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 in 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 given previously, 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 of them during its lifetime.

Memory allocation policies

The system default local allocation policy introduced in the "Linux defaults to the local allocation policy" section 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 usefully 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 that is 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 that are 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, see 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 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, so such a query might reveal useful information, if not already known. The following example shows the command issued on a DL580 G7 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 shows that the task policy is default. The other lines indicate the CPUs that the task may run on (all 40 cores; in this case, hyperthreading has been disabled) and the nodes from

which the task may allocate memory (all four). The cpubind and nodebind lines both represent the set of nodes on whose CPUs the task may 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 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 of the virtual memory ranges listed in this example show the same memory policy: prefer:2. For more information about the preferred memory policy mode, see the "Preferred mode" section. The configuration in the example can only occur in one of two circumstances: either the task has set the memory policy of each of its VMAs to this policy, which is unlikely (for more information, see the "VMA memory policy" section), or all of 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 was 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. A VMA memory 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 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 is displayed; otherwise, the task's memory policy is displayed, if any. If neither VMA nor task memory policy exists, the line specifies 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.

Here 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 preferred mode is almost certainly the task's memory policy. Note that 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 16384 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 each 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 has only touched 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

are shown on 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 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 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, or (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. Shared, read-only page cache pages ignore VMA memory policies.

Most tasks run with 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 fill up 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 numactl to apply them to an

application or shared memory area. These 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.

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 preferred mode with an empty set of nodes.

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.

Specify 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 RHEL5, 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 RHEL6, 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 of 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 slows the program down 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.

Bind mode may be specified 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, see the "Using cpusets" section.

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 and (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. (The other policies could result in complete consumption of memory in this case.) So, 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 that using interleave mode as the task memory policy of an application—for example, to balance the usage of some 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 rebuilt any time the node/memory topology of the server changes.

When the server provides a populated System Locality Information Table (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 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 nondefault 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 just 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. But, 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 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 distros is a SLIT value of 20, and this is the case for both RHEL5 and RHEL6. (The upstream kernel now uses a value of 30 as the threshold; this will appear in newer distro 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 distros such as RHEL6 but not in older distros such as RHEL5. 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 that are discussed here:

- DMA zone—Contains the first 16 MB of physical memory. Drivers for adapters that can address only 16 MB of memory request memory 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 previously, 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 a DL580 G7 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 of 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 it 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 1/2 of the total memory or 70% 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 may 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 RHEL6 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 only and should not be affected by this parameter.

RHEL5's default zonelist ordering resembles node ordering in RHEL6. Memory allocations on node 0 consume the Normal and DMA32 zones before going to the next node in the zonelist. Users with long-lived applications that fit comfortably in the available node 0 memory of a server running RHEL5

sometimes discover when the same server runs RHEL6 that the allocations spill over to another node with an accompanying performance drop. Setting `vm.node_zonelist_order` to `node` is a quick way to address this problem. A better long term solution on RHEL6 might be to affinitize the application's memory usage with the techniques described in this document.

Controlling page migration

Overview

As discussed in the "Linux memory policy principles" section, the kernel applies memory allocation policy when a page is allocated. In general, changes in memory policy do not affect the location of pages that are 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 nonprivileged task can migrate only those pages that are referenced by a single page table entry—that is, that are mapped only into this task's address space. A privileged (root) task can specify an additional flag to request that all 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`, see 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 that are moved between cpusets or when resources of a cpuset are changed, see the "Using cpusets" section.)
    - 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`, see the `move_pages(2)` and `numa_move_pages(3)` manpages.

Both `migrate_pages` and `move_pages` require that the calling task 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 distros, including RHEL5 and RHEL6, 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 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 a DL580, 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 of the pages except those on nodes 1 and 3 to nodes 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 nonprivileged user invokes `migratepages`, it migrates only those pages that are 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, see the `migratepages(8)` manpage.

Important

Take caution when migrating tasks that attach to large shared memory areas. These 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 forms a 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 of the program's private (anon) pages are on node 1, as dictated by the default local allocation policy. Some of `libc`'s shared pages reside on node 0 as well as node 1. The test program has mapped a 1 GB (256K 4 KB pages) anonymous segment at `0x7f607f0d5000`; all of the segment's pages are on node 1.

Example: Nonprivileged user

Assume a nonprivileged 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 of 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 task is the only task that had these pages mapped. The evidence for this 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. Nonprivileged 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 nonprivileged migratepages would not have migrated those pages anyway.

Example: Privileged user

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

Using cpusets (privileged users only) to group tasks and constrain 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 RHEL5 and other older Linux distros, was the cpuset pseudo-file system or `cpusetfs`. It is still available in RHEL6 and other recent Linux distros though 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 distro 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, see 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 RHEL6 and other recent Linux distros.

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, note that a `cpusetfs` file system and a `cgroupfs` cpuset 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, RHEL6 provides a richer and more complex interface in the form of the user space commands delivered in the `libcgroup` package. For more information, see the *RHEL6 Resource Management Guide* at www.redhat.com/docs.)

Assigning tasks to a cpuset is a privileged operation. Assigning resources to a cpuset and modifying the behavior of a cpuset also require 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.

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 each other. 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 of 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 just 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 a DL580 G7 on which hyperthreading 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:

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

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, 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 that 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. So, you could create a cpuset that contains the CPUs and memory of all of the nodes of the system, thus overlapping the top-level cpuset and enable page cache and/or slab interleaving for that cpuset. Then, to spread out their memory load, 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 a DL580 G7 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 node 3 will not have sufficient memory to hold all of 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 a DL580 G7, and you enable the `memory_migrate` control for that cpuset. You 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. Now, 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 are 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 were 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 and memory resources with the application's cpuset. How would this be of benefit?

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. 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 use this sharing of 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 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 (on first touch, for example), 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 one must ensure that 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()`.

References and resources

1. Manpages for commands: `taskset(1)`, `numactl(8)`, `sysctl(8)`, `migratepages(8)`
2. Manpages for libraries: `set_mempolicy(2)`, `get_mempolicy(2)`, `mbind(2)`, `sched_setaffinity(2)`, `sched_getaffinity(2)`
3. Manpages containing general information: `numa(7)`, `cpuset(7)`, `proc(5)`
4. Red Hat's *Performance Tuning Guide* for RHEL6: http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/index.html

For more information

The Documentation directory of the Linux source tree contains numerous documents describing NUMA, memory policy, cpusets, kernel boot parameters, and more. You will probably want the documentation tree from a particular Linux distro and version. For Red Hat Enterprise Linux, the kernel documentation directory is installed by the kernel-doc-<*version*> RPM in the following location: `/usr/share/doc/kernel-doc-<version>/`

The Documentation directory is also available in various distros' kernel source packages.

To see the latest documentation for the upstream Linux kernel (this might discuss features not yet present in a distro), see the Documentation directory in the kernel source tree available at the following website:
www.kernel.org

For more information about HP ProLiant servers, see the HP ProLiant Servers website:
hp.com/go/proliant

5900-2187, March 2012