Red Hat Performance Briefs

RED HAT® ENTERPRISE LINUX® 7: OPTIMIZING MEMORY SYSTEM PERFORMANCE

Red Hat Performance Engineering

Version 1.0

November 2014

100 East Davie Street
Raleigh NC 27601USA
Phone: +1 919 754 4950
Phone: 888 733 4281
Fax: +1 919 800-3804

TABLE OF CONTENTS

1. Introduction

The memory subsystem is one of the most critical components of modern server systems. It stores and supplies critical run-time data and instructions to applications and to the operating system. With today's faster processors, distributing physical system memory across multiple CPU sockets greatly increases memory bandwidth by enabling multiple CPUs to simultaneously access memory without contention, improving overall system performance.

However, from the perspective of any given CPU, only a fraction of system memory will be local to that CPU, while the rest of system memory will be remote to the given CPU and must be accessed over some kind of interconnect. Since modern systems have local memory directly connected to each processor while making memory attached to other CPUs accessible—but at reduced performance—the result is that memory access times are “non-uniform.” Practically all modern multi-socket server systems are non-uniform memory access (NUMA) machines. Figure 1 shows a typical four-socket NUMA system design.

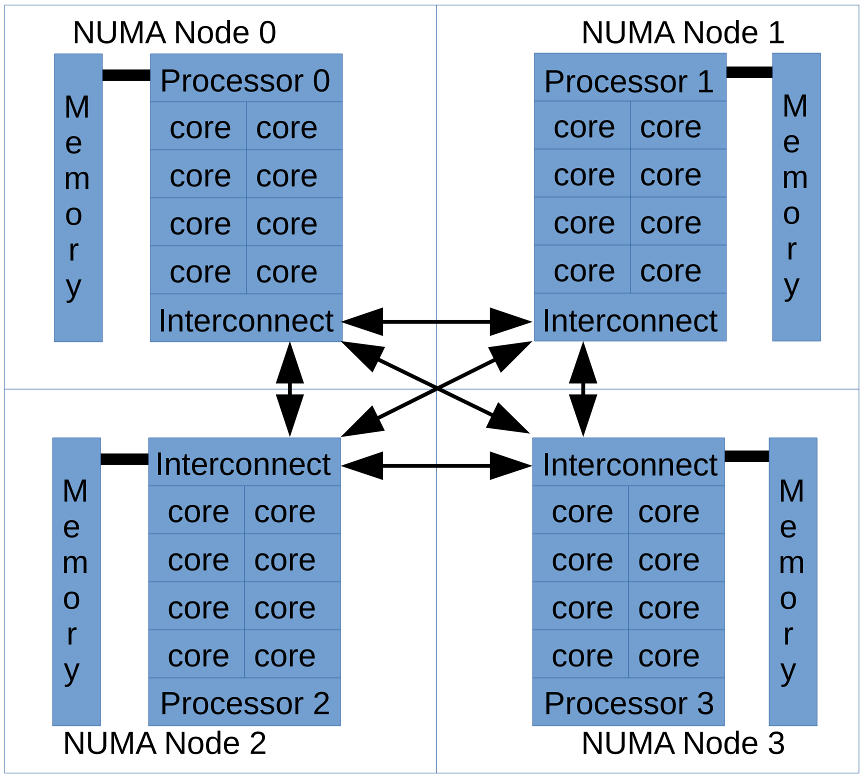

Figure 1: Typical NUMA architecture

Ever-growing demand for memory bandwidth resulting from an increasing number of fast processors, cores, and associated caches found in a typical server system makes effective memory management an important system task. Using local memory is essential for peak performance, and applications will suffer a higher memory latency when accessing data from remote NUMA nodes. System administrators and application users must somehow manage the NUMA characteristics of their servers for optimal application performance.

The good news for users of Red Hat® Enterprise Linux® is that Red Hat has been increasingly automating NUMA memory management optimization tasks with each new release of Red Hat Enterprise Linux. Now, with Red Hat Enterprise Linux 7, most users will get good NUMA system memory management for most applications out of the box. System administrators and application users no longer need to do anything special to achieve good, efficient NUMA system memory management. Read on for background information and details about various NUMA memory management tools, techniques and features available in Red Hat Enterprise Linux.

1.1 What is NUMA?

Practically all modern multi-socket server systems have fractions of system memory distributed among the various CPU sockets. The memory which is local to each socket can be accessed efficiently from the cores in the local CPU. Each CPU socket and its local memory are collectively known as a NUMA node. (More precisely: a NUMA node is a collection of memory and CPU cores which all access the memory with the same latency. Note that some modern systems actually have multiple NUMA nodes on a single CPU socket.)

The NUMA nodes are connected together by one or more buses or interconnects which allow CPUs to access remote memory but at a slower rate. In addition to the higher latency cost of accessing memory one or more hops away from the local CPU, interconnect buses can become bogged down as contended resources if there is significant cross-node memory traffic. (While this paper discusses mostly CPU and memory affinity, it is important to note that I/O devices and associated interrupts can also be local or remote. For applications with a heavy I/O load, remote devices can also significantly bog down node interconnects.)

1.2 Why is NUMA System Memory Management Necessary?

Memory is allocated by default on NUMA nodes containing the cores where application threads are currently executing. Prior to Red Hat Enterprise Linux 7 however, operating system schedulers typically optimized for best overall CPU utilization –rather than for CPU and memory affinity–so subsequent load balancing management might cause execution threads to migrate away from the NUMA nodes where memory was initially allocated. This means that application threads would frequently be scheduled across various CPUs on a NUMA system without consideration for the negative impact on latency associated with accessing data in remote memory. Also, processes with multiple threads might start up with threads running on multiple NUMA nodes, resulting in allocated process memory distributed across the system as shown in the `numastat` output in Figure 2.

```
# numastat -cm qemu

Per-node process memory usage (in MBs)
PID                Node 0 Node 1 Node 2 Node 3  Total
---------------    ------ ------ ------ ------ ------
8249 (qemu-kvm)     12083  13884  12250   6513  44730
8267 (qemu-kvm)      1870  23764  13977   5120  44732
8285 (qemu-kvm)     15751   1486   1841  25649  44726
8303 (qemu-kvm)     11552  10444  18815   3916  44727
---------------    ------ ------ ------ ------ ------
Total               41256  49578  46883  41198 178915

Per-node system memory usage (in MBs):
                   Node 0 Node 1 Node 2 Node 3  Total
                   ------ ------ ------ ------ ------
MemTotal            65416  65536  65536  65536 262024
MemFree             22318  14315  16766  22064  75462
MemUsed             43098  51221  48770  43472 186561
Active              41211  49470  46813  41174 178668
Inactive               46      1      3     12     62
Active(anon)        41209  49469  46810  41164 178652
Inactive(anon)          8      0      0      3     13
Active(file)            2      1      3     10     16
Inactive(file)         37      1      3      8     49
Unevictable             0      0      0      0      0
Mlocked                 0      0      0      0      0
Dirty                   0      0      0      0      0
Writeback               0      0      0      0      0
FilePages              47      2      6     22     78
Mapped                 10      1      3     17     31
AnonPages           41253  49585  46886  41236 178960
Shmem                   8      0      0      3     13
KernelStack             3      1      1      1      5
PageTables             82     96     93     86    356
NFS_Unstable            0      0      0      0      0
Bounce                  0      0      0      0      0
WritebackTmp            0      0      0      0      0
Slab                   32     32     32     55    151
SReclaimable            6     11      8     12     37
SUnreclaim             26     21     24     43    114
AnonHugePages       41238  49570  46842  41180 178830
HugePages_Total         0      0      0      0      0
HugePages_Free          0      0      0      0      0
HugePages_Surp          0      0      0      0      0
```

Figure 2: "numastat -cm qemu" output showing dispersed memory

numastat is a NUMA system memory monitoring tool greatly enhanced by Red Hat with new features released in Red Hat Enterprise Linux 6.4. numastat shows per-NUMA-node memory statistics for processes and the operating system. By default, numastat displays per-node kernel memory allocator hit and miss statistics. Any command-line arguments to numastat will invoke this recently enhanced behavior. In Figure 2, numastat is showing per-NUMA-node memory quantities for processes which match the specified command-line search target pattern "qemu", as well as per-node system memory quantities. numastat makes it clear to see that each of these virtual guests has some memory distributed across all the NUMA nodes. If any given thread needs to access arbitrary process memory, some of the memory accesses will necessarily be remote and the guests will suffer from poor performance due to high memory-access latency. Information provided by numastat has been traditionally used by performance experts to manually place and bind application processes to the most suitable system resources based on a specific server topology.

2. NUMA System Management Tools

There are many other useful system tools and utilities that can be helpful in monitoring and managing a NUMA server environment. One of the most basic but very frequently used tools is lscpu. Figure 3 shows lscpu output from a 4 NUMA node, 40 core system (with hyper-threading disabled). lscpu gives a quick view of the CPU topology: how many sockets, nodes, cores and threads are present in the system. lscpu also provides information about the caches and their sizes, as well as displaying which CPUs are part of which NUMA nodes.

```
# lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                40
On-line CPU(s) list:   0-39
Thread(s) per core:    1
Core(s) per socket:    10
Socket(s):             4
NUMA node(s):          4
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 47
Model name:            Intel(R) Xeon(R) CPU E7- 4870
@ 2.40GHz
Stepping:              2
CPU MHz:               2393.986
BogoMIPS:              4787.86
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              30720K
NUMA node0 CPU(s):     0,4,8,12,16,20,24,28,32,36
NUMA node1 CPU(s):     2,6,10,14,18,22,26,30,34,38
NUMA node2 CPU(s):     1,5,9,13,17,21,25,29,33,37
NUMA node3 CPU(s):     3,7,11,15,19,23,27,31,35,39
```

Figure 3: lscpu output

Figure 4 shows the output from "`numactl --hardware`" (aka "`numactl -H`") for the same system as above. The `numactl` command shows which CPUs are in which NUMA nodes, but also shows the amount of system memory installed and available in each of the NUMA nodes. The bottom of the "`numactl --hardware`" output is the ACPI System Locality Information Table (SLIT) data showing the node distances. The SLIT data reports from the BIOS the expected relative latencies for accessing memory from various NUMA nodes (normalized to 1 and scaled by a factor of 10x). If the row header on the left edge of the table is the NUMA node where a thread is executing, each column to the right in the same row shows the relative cost of accessing memory from that remote NUMA node. Note that accessing memory from the same node where the thread is executing has a relative cost value of 1 (as shown by the diagonal 10 values), but accessing memory from remote nodes on this particular system is expected be 2.1 times slower.

```
# numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 4 8 12 16 20 24 28 32 36
node 0 size: 65415 MB
node 0 free: 22313 MB
node 1 cpus: 2 6 10 14 18 22 26 30 34 38
node 1 size: 65536 MB
node 1 free: 14312 MB
node 2 cpus: 1 5 9 13 17 21 25 29 33 37
node 2 size: 65536 MB
node 2 free: 16755 MB
node 3 cpus: 3 7 11 15 19 23 27 31 35 39
node 3 size: 65536 MB
node 3 free: 22055 MB
node distances:
node   0   1   2   3
  0:  10  21  21  21
  1:  21  10  21  21
  2:  21  21  10  21
  3:  21  21  21  10
```

Figure 4: "numactl --hardware" output

Another excellent tool for visualizing the NUMA topology of a system can be found in the hwloc and hwloc-gui packages. Figure 5 shows the partial output of the `lstopo` command which draws a picture

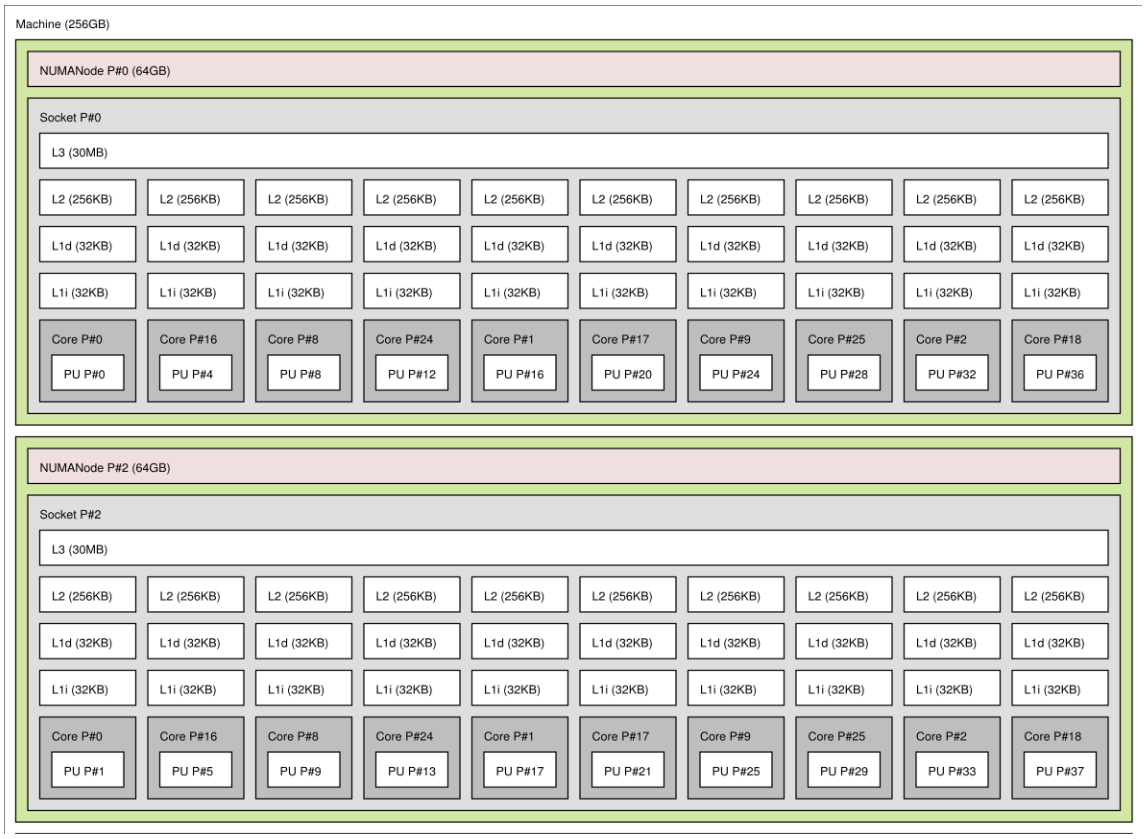

Figure 5: Partial lstopo output (CPUs)

of the system hardware including the cores and caches in each NUMA node.

`top` is an excellent tool for seeing what is running on a system and what kind of resources various processes are consuming. In Red Hat Enterprise Linux 7, there are summary area interactive commands that make it easier to see which NUMA nodes are particularly busy. Pressing '1' while running `top` will show a per-CPU breakdown of utilization statistics. See Figure 6 for an example of `top` per-CPU output.

```
top - 14:50:23 up 23:47,  1 user,  load average: 15.24, 6.99, 2.76
Tasks: 556 total,   1 running, 555 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.3 us,  0.0 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  :  0.0 us, 61.9 sy,  0.0 ni, 38.1 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu8  :  0.3 us,  0.0 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu9  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu10 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu11 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu12 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu13 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu14 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu15 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu16 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu17 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu18 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu19 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu20 :  0.0 us,  0.3 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu21 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu22 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu23 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu24 :  0.3 us,  0.0 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu25 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu26 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu27 :  0.0 us,  0.3 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu28 :  0.0 us,  0.0 sy,  0.0 ni, 99.7 id,  0.3 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu29 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu30 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu31 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu32 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu33 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu34 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu35 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu36 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu37 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu38 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu39 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  26384816+total, 18723772+used, 76610440 free,    28372 buffers
KiB Swap: 21268912+total,        0 used, 21268912+free.    86676 cached Mem
```

Figure 6: top per-cpu output <1>

In Red Hat Enterprise Linux 7, pressing '2' while running `top` will show the utilization statistics on a per-NUMA-node basis as shown in Figure 7.

```
top - 14:51:02 up 23:47,  1 user,  load average: 15.63, 8.00, 3.26
Tasks: 556 total,   1 running, 555 sleeping,   0 stopped,   0 zombie
%Cpu(s): 40.1 us,  0.2 sy,  0.0 ni, 59.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Node0 :  0.1 us,  0.1 sy,  0.0 ni, 99.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Node1 : 80.1 us,  0.0 sy,  0.0 ni, 19.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Node2 :  0.0 us,  0.5 sy,  0.0 ni, 99.5 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Node3 : 80.2 us,  0.0 sy,  0.0 ni, 19.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  26384816+total, 18723627+used, 76611888 free,    28404 buffers
KiB Swap: 21268912+total,        0 used, 21268912+free.    86676 cached Mem
```

Figure 7: top per-NUMA-node output <2>

It is also possible to highlight exclusively the activity of a specified NUMA node in the summary area. Pressing '3' while running `top` will invite selection of a NUMA node to highlight, and the summary area display will look like Figure 8.

```
top - 15:05:08 up 1 day, 1 min,  1 user,  load average: 8.07, 3.02, 2.78
Tasks: 556 total,   2 running, 554 sleeping,   0 stopped,   0 zombie
%Node3 : 80.1 us,  0.0 sy,  0.0 ni, 19.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu11 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu15 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu19 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu23 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu27 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu31 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu35 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu39 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  26384816+total, 18723700+used, 76611160 free,    28556 buffers
KiB Swap: 21268912+total,        0 used, 21268912+free.    86708 cached Mem
```

Figure 8: top highlighted NUMA-node output <3>

2.1 Manual NUMA Binding with taskset and numactl

The `taskset` command can be used to get or set the CPU affinity of a process or thread so that the specified execution threads will run only on the permitted CPUs, and the Linux scheduler will never subsequently migrate the threads away from the permitted CPUs. When combined with the default memory-allocation policy of allocating memory on the current node where the thread is executing, `taskset` can be used to manually launch processes on specified CPUs anticipating that memory will be allocated on the same nodes to achieve CPU and memory affinity. This would enable processes to execute with efficient memory latencies. `taskset` can also be used to move execution threads to other CPUs after they are already running, but note that memory previously allocated by that point will remain on the NUMA nodes where it was originally allocated.

Using the `numactl` command to explicitly specify both the CPUs and the memory resources is a better way to manually bind and manage processes on a NUMA machine (rather than using `taskset` to set the CPU affinity, and relying on the default memory allocation policy). As shown in Figure 9, the `numactl` command can be used to show the binding information of the current process. The `numactl` process in Figure 9 is unbound and permitted to use any of the memory and CPU resources on the system.

```
# numactl --show
policy: default
preferred node: current
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39
cpubind: 0 1 2 3
nodebind: 0 1 2 3
membind: 0 1 2 3
```

Figure 9: "numactl --show" of unbound process

The `numactl` command can also be used to launch commands with specific NUMA memory and execution thread alignment. For example, "numactl -m <NODES> -N <NODES> <COMMAND> {arguments ...}" will run <COMMAND> exclusively on the <NODES> specified. The CPU and memory affinity will be explicit for <COMMAND> and inherited by all of its children.

Figure 10 shows how to bind a launching process and guarantee CPU and memory affinity on the specified NUMA nodes. The Figure 10 example does this via an inline shell script that uses "numactl -m <NODE> -N <NODE>" to do the binding and using "`numactl --show`" to show the effect. Note that `cpubind`, `nodebind`, and `membind` all directly reflect the node where the "`numactl --show`" process is bound. The `physcpubind` output shows a list of individual CPU numbers that comprise the cores on that same node.

```
# for i in 0 1 2 3; do echo; echo "Binding to NUMA node $i";
numactl -m$i -N$i numactl --show; done

Binding to NUMA node 0
policy: bind
preferred node: 0
physcpubind: 0 4 8 12 16 20 24 28 32 36
cpubind: 0
nodebind: 0
membind: 0

Binding to NUMA node 1
policy: bind
preferred node: 1
physcpubind: 2 6 10 14 18 22 26 30 34 38
cpubind: 1
nodebind: 1
membind: 1

Binding to NUMA node 2
policy: bind
preferred node: 2
physcpubind: 1 5 9 13 17 21 25 29 33 37
cpubind: 2
nodebind: 2
membind: 2

Binding to NUMA node 3
policy: bind
preferred node: 3
physcpubind: 3 7 11 15 19 23 27 31 35 39
cpubind: 3
nodebind: 3
membind: 3
```

Figure 10: Manual binding with numactl

Using `numactl` to bind processes with explicit NUMA alignment will sometimes lead to superior performance. However, using `numactl` requires detailed architecture knowledge and careful planning for specific server systems. Furthermore, manual binding like this inherently creates a static arrangement, and when workloads or available resources fluctuate, the original binding may no longer be optimal. Hence, manual placement and binding are best suited for well-defined, static workloads. If the application demand on the system changes or additional hardware resources become available, the planning and resource allocation work will need to be redone—making this approach less effective for dynamic, unpredictable workloads often running in today’s datacenters.

2.2 Automatic NUMA Binding with numad (Red Hat Enterprise Linux 6 and 7)

Red Hat created `numad` to automatically improve NUMA system performance in both static and dynamic workload environments. `numad` is an optional user-level daemon that provides process management and placement advice for efficient use of memory and CPUs on systems with NUMA topology. It is implemented as a CPU-and-memory-affinity management daemon that monitors resource-consuming processes and available NUMA resources. `numad` will attempt to place significant processes (like application workloads and KVM guests) in optimal NUMA locations, dynamically making subsequent adjustments as system conditions change. With the `numad` tool, you can achieve processor and memory affinity—and the resulting superior performance—without requiring repeated human interaction.

`numad` is primarily intended for server consolidation environments, where there might be multiple applications or multiple virtual guests running on the same server system. Because `numad` attempts to align both CPU and memory resources used by running processes on the same NUMA node(s), `numad` is most likely to have a positive effect when processes can be localized and isolated in a fractional subset of the system's NUMA nodes. If the entire system is dedicated to a single large application that necessarily consumes the majority of resources on the system, `numad` is unlikely to improve performance because it would be difficult to localize and isolate multiple processes in their own dedicated subsets of the system resources.

Figure 11 illustrates conceptually how `numad` aligns and consolidates process memory to a single NUMA node. The image on the left depicts how memory is allocated prior to running the `numad` utility, in which processes execute and use memory resources across multiple nodes. The image on the right reflects memory allocation after `numad` has aligned the processes with NUMA nodes. As a result of that alignment, the workloads will no longer interfere with each other and will use the local node's memory optimized for latency, thus boosting overall system performance.

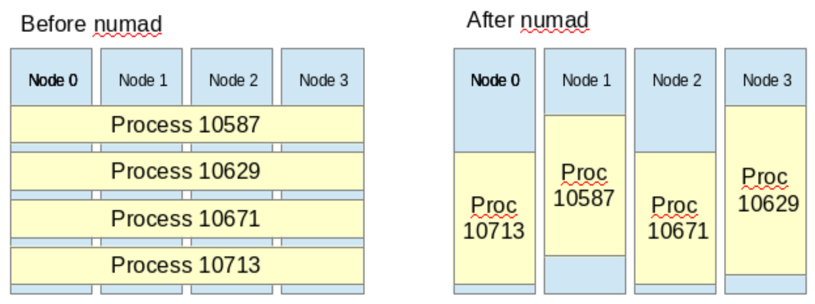

Figure 11: Conceptual affinity alignment

Figure 12 demonstrates the same "`numastat -cm qemu`" command as in Figure 2, however this output is from after `numad` was running on the system. Notice that almost all the memory for each of the four virtual guests running on this system has been localized to a different NUMA node. (Although `numastat` doesn't show it, the KVM guests vCPU threads have also been localized to the same NUMA nodes respectively.) This will optimize the system performance because each virtual guest will effectively have its own resources, and the guests will not interfere with each other.

```
# numastat -cm qemu

Per-node process memory usage (in MBs)
PID              Node 0 Node 1 Node 2 Node 3  Total
---------------  ------ ------ ------ ------ ------
8249 (qemu-kvm)   44750      0      7      0  44757
8267 (qemu-kvm)       0  44727      7      0  44734
8285 (qemu-kvm)       0      0      7  44720  44727
8303 (qemu-kvm)       0      0  44727      0  44727
---------------  ------ ------ ------ ------ ------
Total             44750  44727  44749  44720 178946

Per-node system memory usage (in MBs):
                 Node 0 Node 1 Node 2 Node 3  Total
                 ------ ------ ------ ------ ------
MemTotal          65416  65536  65536  65536 262024
MemFree           18746  19117  18869  18485  75217
MemUsed           46669  46419  46667  47051 186807
Active            44761  44725  44735  44773 178994
Inactive             43      4      7     13     67
Active(anon)      44759  44724  44729  44765 178977
Inactive(anon)        8      0      0      3     13
Active(file)          2      1      6      9     17
Inactive(file)       34      4      7     10     55
Unevictable           0      0      0      0      0
Mlocked               0      0      0      0      0
Dirty                 0      0      0      0      0
Writeback             0      0      0      0      0
FilePages            45      5     13     22     85
Mapped                6      1      9     15     32
AnonPages         44763  44630  44729  44765 178887
Shmem                 8      1      0      3     13
KernelStack           3      1      1      1      5
PageTables           82     96     93     86    357
NFS_Unstable          0      0      0      0      0
Bounce                0      0      0      0      0
WritebackTmp          0      0      0      0      0
Slab                 35     33     34     57    158
SReclaimable          6     10      8     12     37
SUnreclaim           28     23     26     45    121
AnonHugePages      6822    684      0      4   7510
HugePages_Total       0      0      0      0      0
HugePages_Free        0      0      0      0      0
HugePages_Surp        0      0      0      0      0
```

Figure 12: Aligned numastat output

`numad` maintains a specified target utilization level of NUMA node resources while it attempts to localize and isolate processes to particular NUMA nodes. This is intended to preserve a resource margin for additional loads that might need to run. The desired maximum consumption percentage of a NUMA node can be adjusted with the `numad` "-u <percent>" option. The default is 85%. Decrease the target value to maintain more available resource margin on each node. Increase the target value to more exhaustively consume node resources. If you have sized your workloads to precisely fit inside a NUMA node, specifying "-u 100" might improve system performance by telling `numad` to go ahead and consume all the resources in each node.

Another `numad` option that can significantly influence performance is "-H <THP_scan_sleep_ms>". By default, `numad` changes the transparent hugepage scan interval from 10,000ms to 1,000ms. This can be important for applications that use transparent hugepages because hugepages must be reconstructed after memory contents is moved between NUMA nodes. For some workloads, it might be helpful for the hugepage daemon to be even more aggressive when memory moves between nodes. For example, setting this value to 100ms (-H

100) might improve the performance of some workloads which use many transparent hugepages. (Specifying (-H 0) will cause `numad` to retain the system default value.)

Due to dynamic load variations in long-running applications, `numad` is more likely to have a positive effect on system performance when it runs continuously as a daemon. However, `numad` could also be run on demand to isolate workloads and determine NUMA nodes where they should be bound. Moreover, `numad` has an interface that can be queried by various management applications like the virtualization management tool `libvirt`. `numad` provides pre-placement guidance to assist management applications with initial manual binding of CPU and memory resources for their processes.

Using this interface, management applications can receive guidance about the optimal initial binding of CPU and memory resources for the processes that are managed. For details about how to use this pre-placement service, see the `numad` reference page and read about the "-w" option. Figure 13 demonstrates the effect of good workload placement by using the `numad` "-w" option in a psuedo-benchmark that places simulated virtual guest "pig" processes. Note that the average

```
# ./pig_place_test.sh 6 5 8
Trying 6 fake 'guests' with 5 VCPUs and 8 GB each.
Note average work accomplished -- displayed in a few minutes.
numad advises to use nodes: 1 -- but ignoring that and not binding.
../pig_tool/pig -t 5 -Gm 8000 -s 60 -l mem
numad advises to use nodes: 1 -- but ignoring that and not binding.
../pig_tool/pig -t 5 -Gm 8000 -s 60 -l mem
numad advises to use nodes: 7 -- but ignoring that and not binding.
../pig_tool/pig -t 5 -Gm 8000 -s 60 -l mem
numad advises to use nodes: 7 -- but ignoring that and not binding.
../pig_tool/pig -t 5 -Gm 8000 -s 60 -l mem
numad advises to use nodes: 7 -- but ignoring that and not binding.
../pig_tool/pig -t 5 -Gm 8000 -s 60 -l mem
numad advises to use nodes: 7 -- but ignoring that and not binding.
../pig_tool/pig -t 5 -Gm 8000 -s 60 -l mem
Sleeping while the fake guests finish up...
Threads:   5  Avg:  21.2  Stddev:  10.0  Min: 15  Max: 41
Threads:   5  Avg:  23.8  Stddev:   9.8  Min: 16  Max: 42
Threads:   5  Avg:  33.2  Stddev:   8.6  Min: 16  Max: 38
Threads:   5  Avg:  29.4  Stddev:  11.8  Min: 15  Max: 39
Threads:   5  Avg:  19.2  Stddev:   3.1  Min: 16  Max: 25
Threads:   5  Avg:  18.2  Stddev:   2.0  Min: 16  Max: 21

OK, now trying same size fake 'guests' using numad placement advice.
Average work accomplished should be higher, stddev might be better too.
numad advises to use nodes: 1
numactl -N 1 -m 1 ../pig_tool/pig -t 5 -Gm 8000 -s 60 -l mem
numad advises to use nodes: 3
numactl -N 3 -m 3 ../pig_tool/pig -t 5 -Gm 8000 -s 60 -l mem
numad advises to use nodes: 7
numactl -N 7 -m 7 ../pig_tool/pig -t 5 -Gm 8000 -s 60 -l mem
numad advises to use nodes: 4
numactl -N 4 -m 4 ../pig_tool/pig -t 5 -Gm 8000 -s 60 -l mem
numad advises to use nodes: 6
numactl -N 6 -m 6 ../pig_tool/pig -t 5 -Gm 8000 -s 60 -l mem
numad advises to use nodes: 5
numactl -N 5 -m 5 ../pig_tool/pig -t 5 -Gm 8000 -s 60 -l mem
Sleeping while the fake guests finish up...
Threads:   5  Avg:  44.0  Stddev:   0.0  Min: 44  Max: 44
Threads:   5  Avg:  45.0  Stddev:   0.0  Min: 45  Max: 45
Threads:   5  Avg:  44.0  Stddev:   0.0  Min: 44  Max: 44
Threads:   5  Avg:  44.0  Stddev:   0.0  Min: 44  Max: 44
Threads:   5  Avg:  44.0  Stddev:   0.0  Min: 44  Max: 44
Threads:   5  Avg:  45.0  Stddev:   0.0  Min: 45  Max: 45
```

Figure 13: pig pre-placement output

work accomplished in this pseudo-benchmark almost doubles, and the standard deviation of the amount accomplished becomes a non-issue. Good NUMA job placement yields consistent high performance!

`libvirt` will use the `numad` pre-placement advice feature if the "auto" placement parameters are specified in the virtual guest's XML as seen in Figure 14.

The `vcpu` XML element specifies the maximum number of virtual CPUs allocated for the virtual guest. The optional XML attribute placement can be used to indicate the CPU placement mode for the virtual guest. If "placement='auto'" is specified, the virtual guest vcpus will be pinned to the set of nodes returned by querying `numad` for pre-placement advice. The optional `numatune` XML element specifies the NUMA memory strategy `libvirt` should use when starting the virtual guest on a NUMA system; specifying where and how `libvirt` should allocate memory for the guest on a NUMA host. When the "placement='auto'" attribute is set, `libvirt` will limit the virtual guest to using memory only from the set of nodes returned by querying `numad` for pre-placement advice. If placement of `vcpu` is 'auto', and `numatune` is not specified, a default `numatune` with placement 'auto' and mode 'strict' will be added implicitly.

```
<domain>
    ...
    <vcpu placement='auto'>2</vcpu>
    <numatune>
        <memory mode='strict' placement='auto'/>
    </numatune>
    ...
</domain>
```

Figure 14: Virtual Guest auto-placement XML

2.3 Kernel Automatic NUMA Balancing (Red Hat Enterprise Linux 7)

Although `numad` can do a good job providing NUMA process management and placement advice for efficient use of CPU and memory resources on NUMA systems, it is far from perfect. As a user-level daemon, `numad` is limited to somewhat coarse-grained process management. It moves entire processes, and does not know which execution threads are accessing which parts of process memory. `numad` has no knowledge of internal kernel data structures or kernel scheduler details that might facilitate better NUMA process management. Fortunately, the great news for Red Hat Enterprise Linux 7 users is that Red Hat is now building automatic NUMA balancing inside the kernel! This enables much more sophisticated NUMA management even in subtle situations where `numad` might be ineffective.

In Red Hat Enterprise Linux 7, the kernel monitors remote memory accesses and implements multiple strategies to eliminate them. The kernel tracks memory locations by periodically unmapping different parts of process memory and intercepting relatively low-cost page faults to see where the memory is located. This sampling technique has the important benefit that only memory which is actively being used is considered for NUMA remediation. After all, memory which is not being accessed has no memory latency at all.

If NUMA location changes are warranted to improve NUMA efficiency, the scheduler can move execution threads to NUMA nodes with the memory being used by the threads, and it can also move memory to the NUMA nodes where consumer execution threads are executing. Obviously the in-kernel NUMA code has full access to all the internal kernel data structures and knowledge about task scheduling status. It recognizes whether memory is shared or private. If the NUMA balancing feature determines it is necessary to move the memory contents, an efficient "lazy" page migration is utilized that moves memory contents in the background at page fault time.

Already the in-kernel automatic NUMA balancing has achieved excellent results, but Red Hat continues to actively improve the automatic NUMA features. Current improvements underway include changes to better handle large processes and KVM guests which necessarily span more than one NUMA node. The automatic NUMA balancing feature can be turned off by using `sysctl` to disable “kernel.numa_balancing”, but it is enabled and active by default. Red Hat expects most NUMA workloads will perform very well out-of-the-box on Red Hat Enterprise Linux 7 systems without any additional action by users or administrators.

2.4 Performance Comparison of numactl, numad, Automatic NUMA Balancing, no-NUMA

Figure 15 illustrates results of various NUMA management techniques on a Java workload executed on x86 hardware. The dark blue line at the bottom shows the throughput when no NUMA management is used at all. The yellow line—which is mostly hidden under other lines—shows the significant improvement that Red Hat Enterprise Linux 7 brings by default with automatic NUMA balancing. No user or administrator effort is necessary at all, and the automatic NUMA balancing feature achieves almost optimal results for this workload on this system. (Note the system is oversubscribed after 10 warehouses, so most of the results tend to fall off after that point.) The default, out-of-the-box results on Red Hat Enterprise Linux 7 are now good enough that most users with similar loads should just run their workload and let the kernel automatic NUMA balancing functionality optimize the NUMA system performance.

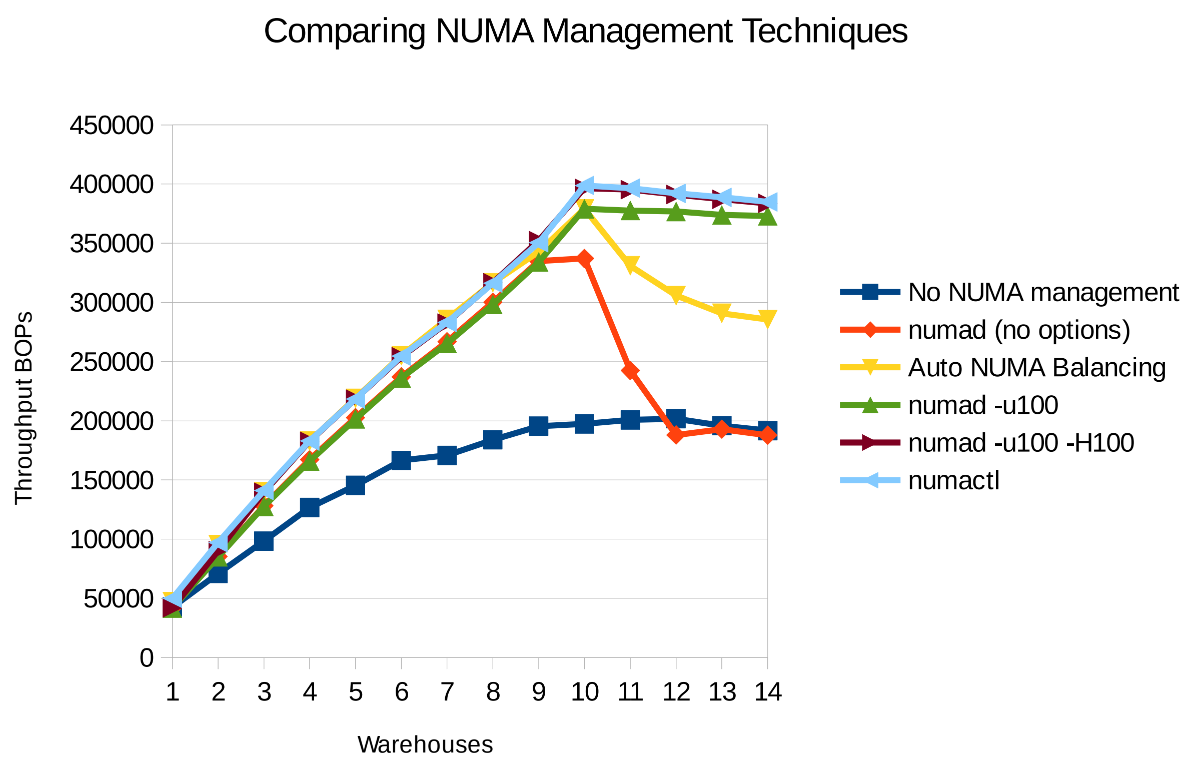

Figure 15: Results of various NUMA management techniques

The light-blue line shows what can be achieved by an expert performance engineer using static manual binding via the `numactl` command. The brown line (mostly hidden under the light-blue line), the green line, and the red line show `numad` results with different options. The red line on the graph shows the performance with `numad` running with no options specified. It starts to fall off after 9 warehouses because `numad` tries to preserve an available resource margin when the utilization reaches 85%. Default `numad` results rapidly degrade in this case when the CPUs are oversubscribed. If the user specifies that `numad` should aim for 100% utilization—shown by the green line—`numad` achieves about the same peak performance improvement as the kernel automatic NUMA balancing functionality does for 10 warehouses. (Note the kernel automatic NUMA performance is better for less than 10 warehouses, where the yellow line is actually under the light-blue line). The "numad -u100" performance degrades much more slowly as resources are oversubscribed. This is shown by the green line being relatively flat towards the right edge of the graph.

For absolute best performance of this particular workload on this particular system, one should either manually bind the processes with `numactl`, or use `numad` specifying both "-u 100" and "-H 100". (The "-H" option sets the khugepaged scan_sleep_millisecs to 100 in this case. Because `numad` causes memory to move from one node to another, it is important to quickly rebuild transparent huge pages.) Since this particular workload is a relatively straight-forward NUMA test, the gross process movements initiated by `numad` are able to achieve the same peak performance improvement accrued by using manual `numad` binding in this case. Using the automatic NUMA balancing features in Red Hat Enterprise Linux 7 is much easier, gives nearly optimal results, and will also work with more subtle and more complicated workloads.

3. General NUMA Guidance

It is a very good idea to size workload processes and virtual guests so they can fit in a single NUMA node whenever possible. This makes it possible for all execution threads in the process to have the lowest possible memory-access latency, and it greatly simplifies optimal NUMA-sensitive placement on the server system. f larger guests or processes are necessary, consider sizing them for even multiples of the resources in NUMA nodes.

For best performance, do not oversubscribe the CPUs or memory on the server system. In many cases it is fine to oversubscribe the resources on the system, but you will not achieve peak performance for individual jobs, especially if many are simultaneously active. The KSM daemon can help with memory oversubscription, but it has additional performance costs of its own. If you must use the KSM daemon on a NUMA system, consider disabling the merge_across_nodes tunable to get better NUMA sensitive performance with KSM.

4. Conclusion

Because almost all modern servers are NUMA systems, Red Hat is continually improving and automating NUMA management tools and features. The NUMA tools in Red Hat Enterprise Linux 6 and 7 help organizations improve application performance on modern hardware systems. For absolute peak performance, expert NUMA tuning might still be useful. But now, in Red Hat Enterprise Linux 7, automatic NUMA balancing should satisfy with the out-of-the-box performance. In many cases the default performance will be near optimal.

4.1 Next Steps

If you are looking for ways to simplify day-to-day management tasks in your current IT infrastructure or are experiencing exponential datacenter growth, Red Hat can help you with new tools, award-winning support, training, and consulting services. Learn more about why Red Hat is the world's most trusted provider of Linux and open source technology today. Visit www.redhat.com.