DOI:10.1145/2500468.2500477

Article development led by acmqueue
queue.acm.org

NUMA becomes more common because memory controllers get close to execution units on microprocessors.

BY CHRISTOPH LAMETER

An Overview of Non-Uniform Memory Access

NON-UNIFORM MEMORY ACCESS (NUMA) is the phenomenon that memory at various points in the address space of a processor have different performance characteristics. At current processor speeds, the signal path length from the processor to memory plays a significant role. Increased signal path length not only increases latency to memory but also quickly becomes a throughput bottleneck if the signal path is shared by multiple processors. The performance differences to memory were noticeable first on large-scale systems where data paths were spanning across motherboards or chassis. These systems required modified operating-system kernels with NUMA support that explicitly understood the topological properties of the system's memory (such as the chassis in which a region of memory was located) in order to avoid excessively long signal path lengths. (Altix and UV, SGI's large address space systems, are examples. These products had to modify the Linux kernel to support NUMA; in these machines, processors in multiple chassis are linked via a proprietary interconnect called NUMALINK).

Today, processors are so fast they usually require memory to be directly attached to the socket they are on. A memory access from one socket to memory from another has additional latency overhead to accessing local memory—it requires the traversal of the memory interconnect first. On the other hand, accesses from a single processor to local memory not only have lower latency compared to remote memory accesses but also do not cause contention on the interconnect and the remote memory controllers. It is good to avoid remote memory accesses. Proper placement of data will increase the overall bandwidth and the latency to memory.

As the trend toward improving system performance by bringing memory even nearer to processor cores continues, NUMA will play an increasingly important role in system performance. Modern processors have multiple memory ports, and the latency of access to memory varies even only depending on the position of the core on the die relative to the controller. Future generations of processors will have increasing differences in performance as more cores on chip necessitate more sophisticated caching. As the access properties of these different kinds of memory continue to diverge, new functionality may be needed in operating systems to allow for good performance.

NUMA systems today are mostly encountered on multisocket systems. A typical high-end business-class server today comes with two sockets and will therefore have two NUMA nodes. Latency for a memory access (random access) is about 100ns. Access to memory on a remote node adds another 50% to that number.

Performance-sensitive applications can require complex logic to handle memory with diverging performance characteristics. If a developer requires explicit control of the placement of memory for performance reasons, some operating systems provide APIs for this (for example, Linux, Solaris, and Microsoft Windows provide system calls for NUMA). However, various heuristics have been developed in the operating systems that manage memory access to allow applications to transparently utilize the NUMA characteristics of the underlying hardware.

A NUMA system classifies memory into *NUMA nodes* (what Solaris calls *locality groups*). All memory available in one node has the same access characteristics for a particular processor. Nodes have an affinity to processors and to devices. These are the devices that can use memory on a NUMA node with the best performance since they are locally attached. Memory is called *node local* if it was allocated from the NUMA node that is best for the processor. For the example, the NUMA system exhibited in Figure 1 has one node belonging to one socket with four cores each.

The process of assigning memory from the NUMA nodes available in the system is called *NUMA placement*. As placement influences only performance and not the correctness of the code, heuristics approaches can yield acceptable performance. In the special case of noncache-coherent NUMA systems, this may not be true since writes may not arrive in the proper sequence in memory. However, noncache-coherent NUMA systems have multiple challenges when attempting to code for them. We restrict ourselves here to the common cache-coherent NUMA systems.

The focus in these discussions will be mostly on Linux since it is an operating system with refined NUMA facilities and is widely used in performance-critical environments today. The author was involved with the creation of the NUMA facilities in Linux and is most familiar with those.

Solaris also has somewhat comparable features,[a] but the number of systems deployed is orders of magnitude less. Work is under way to add support to other Unix-like operating systems, but that support so far has been mostly confined to operating-system tuning parameters for placing memory accesses. Microsoft Windows also has a developed NUMA subsystem that allows placing memory structures effectively, but the software is used mostly for enterprise applications rather than high-performance computing. Requirements on memory-access speeds for enterprise-class applications are frequently more relaxed than in high-performance computing, meaning that less effort is spent on NUMA memory handling in Windows compared with Linux.

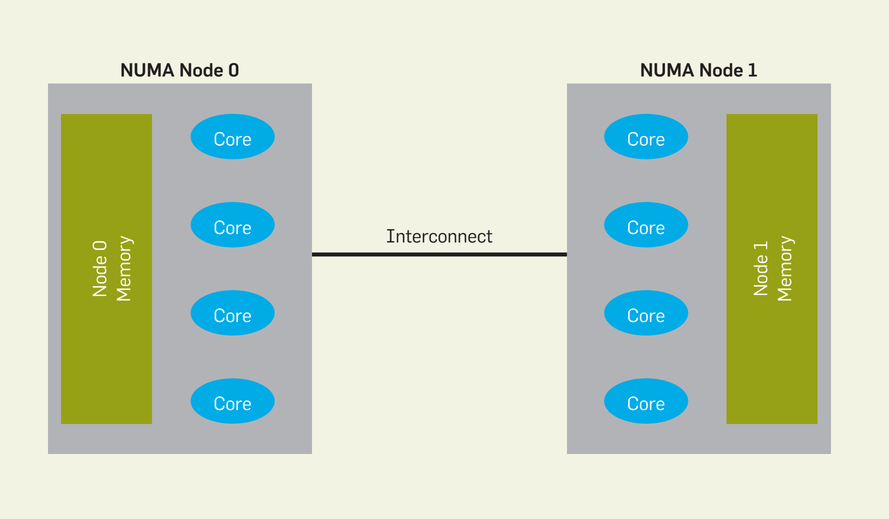
How Operating Systems Handle NUMA Memory

There are several broad categories in which modern production operating systems allow for the management of NUMA: accepting the performance mismatch, hardware memory striping, heuristic memory placement, a static NUMA configurations, and application-controlled NUMA placement.

Ignore the difference. Since NUMA placement is a best-effort approach, one option is simply to ignore the possible performance benefit and just treat all memory as if no performance differences exist. This means the operating system is not aware of memory nodes. The system is functional, but performance varies depending on how memory happens to be allocated. The smaller the differences between local and remote accesses, the more viable this option becomes.

This approach allows software and the operating system to run unmodified. Frequently, this is the initial ap-

Figure 1. A system with two NUMA nodes and eight processors.



[a] For details, see http://docs.oracle.com/cd/E19963-01/html/820-1691/gevog.html; http://docs.oracle.com/cd/E19082-01/819-2239/6n4hsf6rf/index.html; http://docs.oracle.com/cd/E19082-01/819-2239/madv.so.1-1/index.html/.

proach for system software when systems with NUMA characteristics are first used. The performance will not be optimal and will likely be different each time the machine and/or application runs, because the allocation of memory to performance-critical segments varies depending on the system configuration and timing effects on boot-up.

Memory striping in hardware. Some machines can set up the mapping from memory addresses to the cache lines in the nodes in such a way that consecutive cache lines in an address space are taken from different memory controllers (interleaving at the cache-line level). As a result, the NUMA effects are averaged out (since structures larger than a cache line will then use cache lines on multiple NUMA nodes). Overall system performance is more deterministic compared with the approach of just ignoring the difference, and the operating system still does not need to know about the difference in memory performance, meaning no NUMA support is needed in the operating system. The danger of overloading a node is reduced since the accesses are spread out among all available NUMA nodes.

The drawback is the interconnect is in constant use. Performance will never be optimal since the striping means that cache lines are frequently accessed from remote NUMA nodes.

Heuristic memory placement for applications. If the operating system is NUMA-aware (under Linux, NUMA must be enabled at compile time and the BIOS or firmware must provide NUMA memory information for the NUMA capabilities to become active; NUMA can be disabled and controlled at runtime with a kernel parameter), then it is useful to have measures that allow applications to allocate memory that minimizes signal path length so that performance is increased. The operating system has to adopt a policy that maximizes performance for as many applications as possible. Most applications run with improved performance using the heuristic approach, especially compared with the approaches discussed earlier.

A NUMA-aware operating system determines memory characteristics from the firmware and can therefore tune its own internal operations to the memory configuration. Such tuning requires coding effort, however, so only performance-critical portions of the operating system tend to get optimized for NUMA affinities, whereas less-performance-critical components tend to continue to operate on the assumption that all memory is equal.

A NUMA-aware operating system determines memory characteristics from the firmware and can therefore tune its own internal operations to the memory configuration.

The most common assumptions made by the operating system are that the application will run on the local node and that memory from the local node is to be preferred. If possible, all memory requested by a process will be allocated from the local node, thereby avoiding the use of the cross-connect. The approach does not work, though, if the number of required processors is higher than the number of hardware contexts available on a socket (then processors on both NUMA nodes must be used); if the application uses more memory than available on a node; or if the application programmer or the scheduler decides to move application threads to processors on a different socket after memory allocation has occurred.

In general, small Unix tools and small applications work very well with this approach. Large applications that make use of a significant percentage of total system memory and of a majority of the processors on the system will often benefit from explicit tuning or software modifications that take advantage of NUMA.

Most Unix-style operating systems support this mode of operation. Notably, FreeBSD and Solaris have optimizations to place memory structures to avoid bottlenecks. FreeBSD can place memory round-robin on multiple nodes so the latencies average out. This allows FreeBSD to work better on systems that cannot do cache-line interleaving on the BIOS or hardware level (Additional NUMA support is planned for FreeBSD 10). Solaris also replicates important kernel data structures per locality group.

Special NUMA configuration for applications. The operating system provides configuration options that allow the operator to tell the operating system that an application should not be run with the default assumptions regarding memory placement. It is possible to establish memory-allocation policies for an application without modifying code.

Command-line tools exist under Linux that can set up policies to determine memory affinities (`taskset`, `numactl`). Solaris has tunable parameters for how the operating system allocates memory from locality groups as well. These are roughly comparable to Linux's process memory-allocation policies.

Application control of NUMA allocations. The application may want fine-grained control of how the operating system handles allocation for each of its memory segments. For that purpose, system calls exist that allow the application to specify which memory region should use which policies for memory allocations.

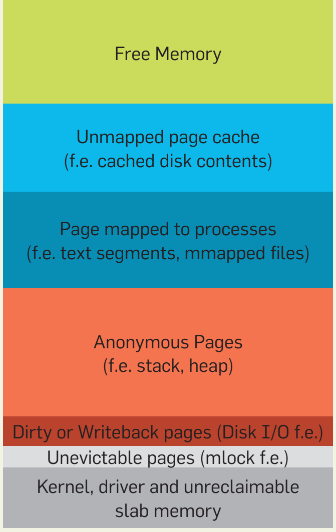
The main performance issues typically involve large structures that are accessed frequently by the threads of the application from all memory nodes and that often contain information that needs to be shared among all threads. These are best placed using interleaving so the objects are distributed over all available nodes.

How Does Linux Handle NUMA?

Linux manages memory in zones. In a non-NUMA Linux system, zones are used to describe memory ranges required to support devices that are not able to perform DMA (direct memory access) to all memory locations. Zones are also used to mark memory for other special needs such as movable memory or memory that requires explicit mappings for access by the kernel (HIGHMEM), but that is not relevant to the discussion here. When NUMA is enabled, then more memory zones are created and they are also associated with NUMA nodes. A NUMA node can have multiple zones since it may be able to serve multiple DMA areas. How Linux has arranged memory can be determined by looking at `/proc/zoneinfo`. The NUMA node association of the zones allows the kernel to make decisions involving the memory latency relative to cores.

On boot-up Linux will detect the organization of memory via the ACPI (Advanced Configuration and Power Interface) tables provided by the firmware and then create zones that map to the NUMA nodes and DMA areas as needed. Memory allocation then occurs from the zones. Should memory in one zone become exhausted, then memory reclaim occurs where Linux will scan through the least recently used pages trying to free a certain number of pages. Counters that show the current status of memory in various nodes/zones can also be seen in `/proc/zoneinfo`. Figure 2 shows types of memory in a zone/node.

Figure 2. Types of memory in a zone/node.


Free Memory
Unmapped page cache
(f.e. cached disk contents)
Page mapped to processes
(f.e. text segments, mmapped files)
Anonymous Pages
(f.e. stack, heap)
Dirty or Writeback pages (Disk I/O f.e.)
Unevictable pages (mlock f.e.)
Kernel, driver and unreclaimable
slab memory

Memory policies. How memory is allocated under NUMA is determined by a memory policy. Policies can be specified for memory ranges in a process's address space, or for a process or the system as a whole. Policies for a process override the system policy, and policies for a specific memory range override a process's policy.

The most important memory policies are:

NODE LOCAL. The allocation occurs from the memory node local to where the code of the process is currently executing.

INTERLEAVE. Allocation occurs round-robin. First a page will be allocated from node 0, then from node 1, then again from node 0, and so on. Interleaving is used to distribute memory accesses for structures that may be accessed from multiple processors in the system in order to have an even load on the interconnect and the memory of each node.

There are other memory policies that are used in special situations that are not mentioned here for brevity's sake. The two policies just mentioned are generally the most useful and are used by default by the operating system. NODE LOCAL is the default allocation policy if the system is up and running.

The Linux kernel will use the INTERLEAVE policy by default on bootup. Kernel structures created during bootstrap are distributed over all the nodes available in order to avoid putting excessive load on a single memory node later when processes require access to the operating-system structures. The system default policy is changed to NODE LOCAL when the first userspace process (`init` daemon) is started.

The active memory allocation policies for all memory segments of a process (and information that shows how much memory was actually allocated from which node) can be seen by determining the process `id` and then looking at the contents of `/proc/<pid>/numa_maps`.

Basic operations on process startup. Processes inherit the memory policy from their parent. Most of the time the policy is left at the default, which means NODE LOCAL. When a process is started on a processor, then memory is allocated for that process from the local NUMA node. All other allocations of the process (through growing the heap, page faults, mmap, and so on) will also be satisfied from the local NUMA node.

The Linux scheduler will attempt to keep the process cache hot during load balancing. This means the preference of the scheduler is to leave the process on processors that share the L1-processor cache, then on processors that share L2, and then on processors that share L3 with the processor the process ran on last. If there is an imbalance beyond that, then the scheduler will move the process to any other processor on the same NUMA node.

As a last resort the scheduler will move the process to another NUMA node. At that point the code will be executing on the processor of one node, while the memory allocated before the move will have been allocated on the old node. Most memory accesses from the process will then be remote, which will cause the performance of the process to degrade.

There has been some recent work in making the scheduler NUMA-aware to ensure the pages of a process can be moved back to the local node, but that work is available only in Linux 3.8 and later, and is not considered mature yet. Further information on the state of affairs may be found on the Linux kernel mailing lists and in articles on http://lwn.net.

Reclaim. Linux typically allocates all available memory in order to cache data that may be used again later. When memory begins to be low, reclaim will be used to find pages that are either not in use or unlikely to be used soon. The effort required to evict a page from memory and to get the page back if the need arises varies by type of page. Linux prefers to evict pages from disk that are not mapped into any process space because it is easy to drop all references to the page. The page can be reread from disk if required later. Pages that are mapped into a process's address space require the page first be removed from that address space before the page can be reused. A page that is not a copy of a page from disk (anonymous pages) can be evicted only if the page is first written out to swap space (an expensive operation). There are also pages that cannot be evicted at all, such as `mlocked()` memory or pages in use for kernel data.

The impact of reclaim on the system can therefore vary. In a NUMA system there will be multiple types of memory allocated on each node. The amount of currently free space on each node will vary. So if there is a request for memory and the local node would require reclaim but another node has enough memory to satisfy the request without reclaim, then the kernel has two choices:

- Run a reclaim pass on the local node (causing kernel processing overhead) and then allocate node-local memory to the process.
- Just allocate from another node that does not need a reclaim pass. Memory will not be node local, but we avoid frequent reclaim passes. Reclaim will be performed when all zones are low on free memory. This approach reduces the frequency of reclaim and allows more of the reclaim work to be done in a single pass.

For small NUMA systems (such as the typical two-node servers) the kernel defaults to the second approach. For larger NUMA systems (four nodes and higher) the kernel will perform a reclaim in order to get node-local memory whenever possible because the latencies have higher impacts on process performance.

There is a knob in the kernel that determines how the situation is to be treated in `/proc/sys/vm/zone_reclaim`. A value of 0 means that no local reclaim should take place. A value of 1 tells the kernel that a reclaim pass should be run in order to avoid allocations from the other node. On boot-up a mode is chosen based on the largest NUMA distance in the system.

If zone reclaim is switched on, then the kernel still attempts to keep the reclaim pass as lightweight as possible. By default, reclaim will be restricted to unmapped page-cache pages. The frequency of reclaim passes can be further reduced by setting `/proc/sys/vm/min_unmapped_ratio` to the percentage of memory that must contain unmapped pages in order to run a reclaim pass. The default is 1%.

Zone reclaim can be made more aggressive by enabling write-back of dirty pages or the swapping of anonymous pages, but in practice doing so has often resulted in significant performance issues with reclaim.

Basic NUMA command-line tools. The main tool used to set up the NUMA execution environment for a process is `numactl`, which also allows the display of the system NUMA configuration, as well as the control of shared memory segments. It is possible to restrict processes to a set of processors, as well as to a set of memory nodes. `Numactl` can be used, for example, to avoid task migration between nodes or restrict the memory allocation to a certain node. Note that additional reclaim passes may be required by the kernel if the allocation is restricted. Those cases are not influenced by zone-reclaim mode because the allocation is restricted by a memory policy to a specific set of nodes, and therefore the kernel does not have a choice simply to pick memory from another NUMA node.

Another tool that is frequently used for NUMA is `taskset`. It basically allows only binding of a task to processors and therefore has only a subset of `numactl`'s capability. `Taskset` is heavily used in non-NUMA environments, and therefore the familiarity results in developers preferring to use `taskset` instead of `numactl` on NUMA systems.

NUMA information. There are numerous ways to view information about the NUMA characteristics of the system and of various processes currently running. The hardware NUMA configuration of a system can be viewed through the use of `numactl --hardware`. This includes a dump of the SLIT (system locality information table) that shows the cost of accesses to different nodes in a NUMA system. The example in Figure 3 shows a NUMA system with two nodes. The distance for a local access is 10. A remote access costs twice as much on this system (20). This is the

Figure 3. Displaying NUMA characteristics of a system.

```
$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
node 0 size: 131026MB
node 0 free: 588MB
node 1 cpus: 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31
node 1 size: 131072MB
node 1 free: 169MB
node distances:
node   0   1
  0:  10  20
  1:  20  10
```

convention, but the practice of some vendors (especially for two-node systems) is simply to provide 10 and 20 without regard to the actual latency differences to memory.

`Numastat` is another tool that is used to show how many allocations were satisfied from the local node. Of particular interest is the `numa_miss` counter, which indicates the system assigned memory from a different node in order to avoid reclaim. These allocations also contribute to other node. The remainder of the count are intentional off-node allocations. The amount of off-node memory can be used as a guide to figure out how effectively memory was assigned to processes running on the system (see Figure 4).

How memory is allocated to a process can be seen via a status file in `/pro/<pid>/numa_maps` (illustrated in Figure 5).

The output shows the virtual address of the policy and then some information about what the NUMA characteristics are of the memory range. `Anon` means the pages do not have an associated file on disk. `Nx` shows the number of pages on the respective node.

The information about how memory is used in the system as a whole is available in `/proc/meminfo`. The same information is also available for each NUMA node in `/sys/devices/system/node/node<X>/meminfo`. Numerous other bits of information are available from the directory where `meminfo` is located. It is possible to compact memory, get distance tables, and manage huge pages and mlocked pages by inspecting and writing values to key files in that directory.

First-touch policy. Specifying memory policies for a process or an address range does *not* cause any allocation of memory, which is often confusing to newcomers. Memory policies specify what should happen *when* the system needs to allocate memory for a virtual address. Pages in a process's memory space that have not been touched or that are zero do not have memory assigned to them. The processor will generate a hardware fault when a process touches or writes to an address (*page fault*) that is not populated yet. During page-fault handling by the kernel, the page is allocated. The instruction that caused the fault is then restarted and will be able to access the memory as needed.

What matters, therefore, is the memory policy in effect *when the allocation occurs*. This is called the *first touch*. The first-touch policy refers to the fact that a page is allocated based on the effective policy when some process first uses a page in some fashion.

Figure 4. Displaying NUMA statistics of the system.

```
$ numastat
                node0        node1
numa_hit        13273229839  4595119371
numa_miss       2104327350   6833844068
numa_foreign    6833844068   2104327350
interleave_hit  52991        52864
local_node      13273229554  4595091108
other_node      2104327635   6833872331
```

Figure 5. Displaying NUMA settings and statistics of the system.

```
# cat /proc/1/numa_maps
7f830c175000 default anon=1 dirty=1 active=0 N1=1
7f830c177000 default file=/lib/x86_64-linux-gnu/ld-2.15.so anon=1 dirty=1
active=0 N1=1
7f830c178000 default file=/lib/x86_64-linux-gnu/ld-2.15.so anon=2 dirty=2
active=0 N1=2
7f830c17a000 default file=/sbin/init mapped=18 N1=18
7f830c39f000 default file=/sbin/init anon=2 dirty=2 active=0 N1=2
7f830c3a1000 default file=/sbin/init anon=1 dirty=1 active=0 N1=1
7f830dc56000 default heap anon=223 dirty=223 active=0 N0=52 N1=171
7fffb6395000 default stack anon=5 dirty=5 active=1 N1=5
```

The effective memory policy on a page depends on memory policies assigned to a memory range or on a memory policy associated with a task. If a page is only in use by a single thread, then there is no ambiguity as to which policy will be followed. However, pages are often used by multiple threads. Any one of them may cause the page to be allocated. If the threads have different memory policies, then the page may as a result seem to be allocated in surprising ways for a process that also sees the same page later.

For example, it is fairly common that text segments are shared by all processes that use the same executable. The kernel will use the page from the text segment if it is already in memory regardless of the memory policy set on a range. The first user of a page in a text segment will therefore determine its location. Libraries are frequently shared among binaries, and especially the C library will be used by almost all processes on the system. Many of the most-used pages are therefore allocated during boot-up when the first binaries run that use the C library. The pages will at that point become established on a particular NUMA node and will stay there for the time the system is running.

First-touch phenomena limit the placement control that a process has over its data. If the distance to a text segment has a significant impact on process performance, then dislocated pages will have to be moved in memory. Memory could look like it was allocated on NUMA nodes not permitted by the memory policy of the current task because an earlier task already brought the data into memory.

Moving memory. Linux has the capability to move memory. This means the virtual address of the memory in the process space stays the same. Only the physical location of the data is moved to a different node. The effect can be observed by looking at `/proc/<pid>/numa_maps` before and after a move.

Migrating all of a process's memory to a node can optimize performance of an application by avoiding cross-connect accesses should the system have placed pages on other NUMA nodes. However, a regular user can move only pages of a process that are referenced by only that process alone (otherwise, the user could interfere with performance optimization of processes owned by other users). Only `Root` has the capability to move all pages of a process.

It can be difficult to ensure all pages are local to a process since some text segments are heavily shared and there can be only one page backing an address of a text segment. This is particularly an issue with the C library or other heavily shared libraries.

Linux has a `migratepages` command-line tool to manually move pages around by specifying a `pid`, as well as the source and destination nodes. The memory of the process will be scanned for pages currently allocated on the source node. Those will be moved to the destination node.

NUMA scheduling. The Linux scheduler had no notion of the page placement of memory in a process until Linux 3.8. Decisions about migrating processes were made on an estimate of the cache hotness of a process's memory. If the Linux scheduler moved the execution of a process to a different NUMA node, then the performance of that process could be significantly impacted because its memory now would require access via the cross-connect. Once that move was complete the scheduler would estimate the process memory is cache hot on the remote node and leave the process there as long as possible. As a result, administrators who wanted the best performance felt it best not to let the Linux scheduler interfere with memory placement. Processes were often pinned to a specific set of processors using `taskset`, or the system was partitioned using the `cpusets` feature to isolate applications to stay within the NUMA node boundaries.

In Linux 3.8 the first steps were made to address this situation by merging a framework that will enable the scheduler at some point to consider the page placement and perhaps automatically migrate pages from remote nodes to the local node. However, there is a significant development effort still needed, and the existing approaches do not always enhance the performance of a given computing load. This was the state of affairs earlier this year; for more recent information on the Linux kernel mailing list, see http://vger.kernel.org or articles from *Linux Weekly News* (http://lwn.net; for example, http://lwn.net/Articles/486858/).

Conclusion

NUMA support has been around for a while in various operating systems. NUMA support in Linux has been available since early 2000 and is being continually refined. Frequently kernel NUMA support will optimize process execution without the need for user intervention, and in most use cases an operating system can simply be run on a NUMA system, providing decent performance for typical applications.

Special NUMA configuration through tools and kernel configuration comes into play when the heuristics provided by the operating system do not provide satisfactory application performance to the end user. This is typically the case in high-performance computing, high-frequency trading, and for real-time applications, but recently these issues have become more significant for regular enterprise-class applications. Traditionally, NUMA support required special knowledge about the application and hardware for proper tuning using the knobs provided by the operating systems. Recent developments point (especially around the Linux NUMA scheduler) to developments that will result in the ability of the operating systems to automatically balance a NUMA application load properly over time.

The use of NUMA needs to be guided by the increase in performance that is possible. The larger the difference between local and remote memory access, the greater the benefits that arise from NUMA placement. NUMA latency differences are due to memory accesses. If the application does not rely on frequent memory accesses (because, for example, the processor caches absorb most of the memory operations), then NUMA optimizations will have no effect. Also for I/O-bound applications the bottleneck is typically the device and not memory access. An understanding of the characteristics of the hardware and software is required in order to optimize applications using NUMA.

Related articles on queue.acm.org

Photoshop Scalability: Keeping It Simple
Clem Cole and Russell Williams
http://queue.acm.org/detail.cfm?id=1858330

The Cost of Virtualization
Ulrich Drepper
http://queue.acm.org/detail.cfm?id=1348591

Performance Anti-Patterns
Bart Smaalders
http://queue.acm.org/detail.cfm?id=1117403

Additional Reading

Braithwaite, R., McCormick, P., Feng, W.
Empirical memory-access cost models in multicore NUMA architectures. Virginia Tech Department of Computer Science, 2011.

Hacker, G.
Using NUMA on RHEL 6; http://www.redhat.com/summit/2012/pdf/2012-DevDay-Lab-NUMA-Hacker.pdf.

Kleen, A.
***A NUMA API for Linux*. Novell, 2005; http://developer.amd.com/wordpress/media/2012/10/LibNUMA-WP-fv1.pdf.**

Lameter, C.
Effective synchronization on Linux/NUMA systems. Gelato Conference, 2005. Effective synchronization on Linux/NUMA systems. Gelato Conference, 2005.

Lameter, C.
2006. Remote and local memory: Memory in a Linux/NUMA system. Gelato Conference, 2006.

Li, Y., Pandis, I., Mueller, R., Raman, V., Lohman, G.
NUMA-aware algorithms: The case of data shuffling. University of Wisconsin-Madison / IBM Almaden Research Center, 2013.

Love, R.
2004. Linux Kernel Development. Indianapolis: Sams Publishing.

Oracle.
Memory and Thread Placement Optimization Developer's Guide, 2010; http://docs.oracle.com/cd/E19963-01/html/820-1691/.

Schimmel, K.
Unix Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers. Addison-Wesley, 1994.

Christoph Lameter specializes in high-performance computing and high-frequency trading technologies. As an operating-system designer and developer, he has been developing memory management technologies for Linux to enhance performance and reduce latencies. He is fond of new technologies and new ways of thinking that disrupt existing industries and cause new development communities to emerge.