

NUMA aware heap memory manager

By Patryk Kaminski

Patryk.Kaminski@amd.com

Introduction

Writing software for multi-processor systems is not an easy task. In an ideal scenario, as the total number of processors increase in a system, the throughput of an application should also scale proportionally. However, this is rarely the case. Thread synchronization and accessing shared resources can cause the code to execute serially, and possibly produce bottlenecks. For example, when multiple processors use the same bus to access the memory, the bus can become saturated. As the number of processors in the system increases, the available memory bandwidth to each CPU decreases. Ideally, doubling the number of processors should double the performance, but this is almost never the case. In fact, in many scenarios, increasing the number of processors in the system may cause performance degradation.

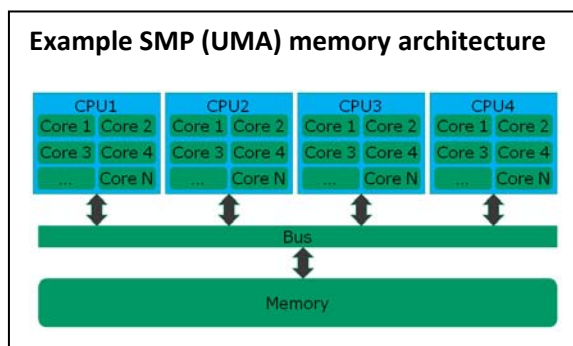
One approach for reducing the shared memory bus bottleneck is through the use of Non-Uniform Memory Access (NUMA) system architecture. In the NUMA architecture, each processor may have its own local memory. There may also be a mechanism allowing one processor to access memory connected to another processor. Typically a processor can access the local memory (connected directly to the processor) faster than the remote memory (connected to another processor). The main challenge with NUMA is controlling where the memory for data and code is allocated. However, carefully managing memory can be an added burden for programmers. Implementing effective software solutions can be a very challenging task for a number of technical reasons, which is why many real-world software applications simply choose to ignore the problem.

One possible solution is to implement a software library, which provides a generalized memory management solution that takes advantage of the NUMA architecture. This approach helps eliminate or reduce performance bottlenecks caused by memory bus congestion or remote memory accesses. This article investigates an attempt to come up with such a solution.

System Memory Architectures

Uniform Memory Access (UMA)

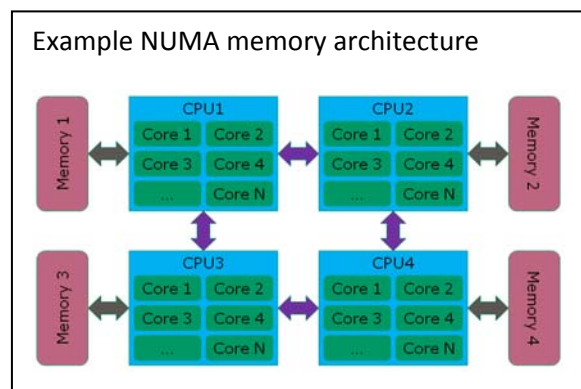
Traditional X86 systems were based on Uniform Memory Access (UMA) shared memory architecture. The most commonly used UMA architecture was the bus-based symmetric



multiprocessing (SMP) architecture where multiple CPUs access the memory via a shared bus. The memory access time for any CPU is the same but the shared memory bus can become a major bottleneck. CPU manufacturers attempt to mitigate the bottleneck by increasing the CPU cache sizes. Large caches increase the chance that the CPU will find the data it needs in the local cache and may not have to access memory at all. Unfortunately, a large cache is not a general solution to the memory bottleneck problem. There are memory intensive applications that frequently use large areas of memory that doesn't fit in the available cache. In such cases the memory bottleneck problem remains. Further, the problem increases as the number of CPUs connected to the shared bus increases.

Non-uniform Memory Access (NUMA)

AMD Opteron™ CPUs introduced X86 NUMA architecture based systems. In a typical AMD Opteron based multiprocessor system, each CPU has its own memory. The CPUs also have direct links (HyperTransport™ bus) to each other, allowing one CPU to transparently access memory connected to another CPU. This mechanism also solves the problem of cache coherency, so technically the systems follow a Cache-Coherent NUMA (ccNUMA) architecture.



In the NUMA architecture the memory access time is not constant. In the example above, CPU1 can access its local memory in bank 1 much faster than it can access memory connected to CPU2 in bank 2 (requires one hop over the HT link). Accessing memory connected to CPU4 (bank 4) from CPU 1 has two hops over HT link. One of the main advantages of the NUMA architecture is that it provides a much better scalability for systems with large number of CPUs. However, the increased cost of accessing remote memory over local memory can affect performance, in other words, software can maximize performance by increasing usage of local memory. For example, data stored in memory bank 1 will be best processed by code executed on CPU1, while data stored in memory bank 3 will be best processed by code executed on CPU3.

Well designed, NUMA-aware software carefully allocates memory and manages threads to maximize the local memory usage. However, it is a difficult task to determine the topology of the system, allocate memory from specific memory banks, and ensure that the data is being manipulated by threads running on the local CPU.

Heap memory managers

Operating systems provide multiple APIs for memory allocation and management. Unfortunately, these APIs are not always very efficient. In general, making an OS API call is expensive because of the context switch between user mode and the system kernel. Further, the APIs often have limitations such as large minimum allocation size. For example an API

function may always allocate a whole page (4 KB) of memory even if the caller requested a much smaller size. This poses a serious problem for applications that frequently allocate and release small memory blocks. To solve these problems, programmers typically use heap memory manager libraries. The standard C/C++ libraries for most popular C/C++ compilers include such heap manager implementations, but there are also many other 3rd party options.

A typical heap memory manager uses the OS API to allocate large memory blocks at a time and divides these blocks into smaller parts to satisfy memory requests by the calling program. This reduces the cost of API call overhead. For example, an application may make 1000 calls to allocate 64 bytes of data, but the heap manager may make only a single OS API call to allocate a large memory block (ex. 1 MB), and carve out 64 bytes of memory for each 64 byte request. When the initial pool of 1 MB of memory is used up, the heap manager will typically make another OS API call to allocate more memory.

The heap memory manager can also allow applications to allocate memory blocks of smaller size which may help reduce the waste of memory due to fragmentation.

Lock contention

A heap memory manager must keep track of all the memory that it requested from the operating system. In particular it must track which parts of the memory are free and which have been allocated to the application. In multithreaded environments this task is further complicated as multiple threads request to allocate or release memory from the heap manager at the same time. A typical solution to this is to have a thread synchronization mechanism, ex. a single lock, to prevent memory corruption.

When a thread tries to allocate memory, the lock is acquired, memory is carved out from the free pool, then the lock is released and the requested memory block is returned to the calling code. The problem with this implementation is that it does not scale on systems with a large number of CPUs. To the contrary, if the program performs many memory allocations on multiple threads, all threads will be effectively serialized by waiting on the same lock.

Advanced heap memory managers resolve this problem by using a lock-free approach. They allocate and release memory (at least in some scenarios) without using any locks for synchronization. This alone provides a significant performance boost.

NUMA and Virtual Memory

Modern operating systems use virtual memory and give applications limited control over the mapping of virtual to physical memory. When an application allocates a memory block (using an OS API call or a heap memory manager), it is assigned a virtual memory region. The OS maps that virtual memory region to some physical memory location, but the OS typically retains a full control over when that happens or what physical memory range to use.

Modern operating systems such as Microsoft® Windows® and Linux® use a “first touch” policy. What this means is that when an application requests memory, the virtual address is initially not mapped to any physical memory. When the application first accesses the memory (read or write), the OS allocates a physical memory region and maps the virtual address to the physical range. The OS typically allocates physical memory from the same NUMA node as the CPU that executed the thread which first accessed the virtual memory block.

There are additional tools to help programmers better control the memory allocations and thread execution on NUMA systems. For example Microsoft Windows Vista® provides an API that allows an application to allocate memory on a specific NUMA node. When Windows® allocates physical memory for the memory block, it will try to allocate it on local or a specified NUMA node if possible. Linux provides a library ([libnuma](#)) that provides similar functionality. Unfortunately, both approaches have limitations that can adversely affect performance. Using these APIs means that the programmer cannot use, and may lose, the benefits of heap memory managers provided in the CRT resulting in a high cost for memory allocation/management and potentially high memory fragmentation.

NUMA aware heap memory manager

As explained above, ensuring optimal performance on a NUMA system is a difficult task and cannot be achieved by the operating system alone. The “first-touch” policy improves the probability that a thread will be assigned physical memory from the local NUMA node, but if the application uses a heap memory manager, and later releases the memory, and re-allocates it in a different thread, it is likely that the new thread will be assigned memory that has already been committed to physical memory from a remote NUMA node. This is because typical heap memory managers are not NUMA aware; they do not keep track of memory locality and can interfere with OS efforts to assign local physical memory to each thread.

The only solution to this problem is to have a NUMA aware heap memory manager.

On the surface, the problem may seem easy to solve: since the heap manager uses an OS API to allocate large blocks of memory, such API calls need only to be replaced with NUMA aware equivalents. When a thread running on CPU X tries to allocate memory for the first time, the heap manager will call the NUMA aware API to allocate a large memory block (which will return a memory block on local NUMA node). The heap manager will then carve out a small memory block of the requested size and return it to the caller. This way the thread will be given a memory block of the requested size on a local NUMA node. However, if another thread running on a CPU on a different NUMA node requests memory, the heap manager will simply carve out another memory block from its pool of free memory which is on a different NUMA node. The situation is further complicated if the thread frees the memory (returns it to the heap manager), and the memory is later reused by another thread. For example, if thread A running on CPU X allocated and used a memory block, the corresponding physical memory will be allocated from the same NUMA node as CPU X. If thread A later releases the memory, the heap manager may decide to keep it for later use. If a thread B running on CPU Y tries to allocate memory,

chances are that the heap manager will simply assign it the same memory that was just used by thread A and is already committed to a different NUMA node.

In early 2008 the author implemented a heap memory manager that addressed all of the above problems, that is:

1. Reduce the overhead of OS API calls by managing its own memory pool
2. Reduce the lock contention in multi-processor systems
3. Maximize local memory allocations on NUMA systems

To simplify the task, we decided to start with an existing heap memory manager that already solved problems 1 and 2 above, and modify it to be NUMA aware. After analysis of many heap memory managers it was decided to use TCMalloc heap memory manager as the starting point.

TCMalloc

TCMalloc is an open source project available under the New BSD license. It is part of the google-perftools package and can be downloaded from <http://code.google.com/p/google-perftools/>.

As of this time, TCMalloc is supported on Linux (32 and 64 bit), Windows (32 bit only) and Solaris. It is intended as a complete replacement for the heap manager in the CRT libraries. It provides the entire suite of calls for C (malloc, free, realloc, calloc, etc.) and C++ (new, delete). It is easy to use – programmers simply link the tcmalloc library to their own code; all CRT heap memory calls are automatically substituted with TCMalloc equivalents.

TCMalloc provides a very effective solution to the problem of the high cost of the OS memory API overhead. It typically allocates memory in very large memory blocks, splits them into smaller chunks, and manages them. It has built in mechanism to reduce memory fragmentation. Most importantly, it keeps separate memory pools for each thread that is used for small memory allocations. This greatly reduces the lock contention since TCMalloc can typically allocate small memory objects from the local (per thread) memory pool without any thread synchronization.

Considerations, objectives, OS limitations

An initial goal was to maintain compatibility with Linux and Microsoft Windows XP Windows Vista. All of these operating systems use the first-touch policy for physical memory allocations. On Linux, it is also possible to use [libnuma](#) – an open source library implemented by Andi Kleen for SUSE Linux – that provides a mechanism to set NUMA policies and allocate/manage memory allocations, thread scheduling, etc. Although [libnuma](#) is not available for Microsoft Windows, Windows Vista provides some additional APIs, such as [VirtualAllocExNuma](#), that offer similar functionality. Unfortunately, no such APIs are available for Windows XP. Since the author

wanted to keep the same design on all architectures, NUMA-specific APIs provided by [libnuma](#) or Windows Vista could not be used.

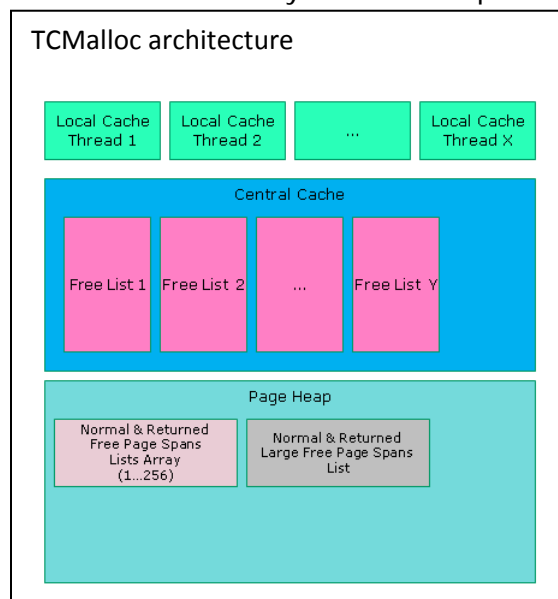
TCMalloc architecture

TC Malloc uses three levels of caching to maximize performance. For each application thread, it creates a local per-thread cache. This is used for small allocations only (32 KB or less) and helps reduce the lock contention because memory allocations from local cache do not require thread synchronization.

The next level is the central cache where TCMalloc keeps lists of free memory blocks of various sizes. TCMalloc keeps a separate free list for each of its 256 predefined object sizes (from 8 bytes to 32 KB) allowing it to quickly find a free memory region of specified size.

The third level is the page heap where it keeps track of large memory blocks allocated using an OS API. The memory objects in the page heap are divided into chunks of one of the 256 predefined sizes (8 bytes to 32 KB).

When a small memory allocation request comes in (32 KB or less,) the heap manager first



checks to see if it can be completed from the local cache. If not, it checks the central cache (at this point it uses lock synchronization). If the central cache does not have a free object of the specified size, it checks the page heap. If the page heap does not have a free memory object of sufficiently large size, TCMalloc uses an OS specific API to allocate another large block of memory (typically 1 MB or more). The memory is then divided into smaller chunks and moved to one of the free lists in the central cache. Next, a portion of free memory objects of the predefined size is moved from the free list to the local cache for the calling thread.

For large memory allocations, TCMalloc bypasses the local and central cache and immediately tries to use the page heap to try to complete the request.

When a thread frees a small memory object, the memory is first placed back in the local cache in the list for the corresponding size. If the free list is too big, TCMalloc transfers some of the free memory blocks back to the central cache. TCMalloc also tries to reduce memory fragmentation by tracking when consecutive memory blocks are freed. When the free list of consecutive memory blocks in the central cache exceeds a specified threshold, TCMalloc moves it back to the page heap. TCMalloc periodically releases the memory blocks in the page heap back to the OS. However, it only releases the underlying physical memory, not the virtual memory. This helps the OS to reclaim the memory and assign it to other processes. Memory

spans for which the physical memory has been released back to the main OS are moved to the returned span list. When looking for a free memory block, TCMalloc first checks the normal span list array, then the returned span list array, and finally it will request more memory using an OS API.

First attempt – with common heap

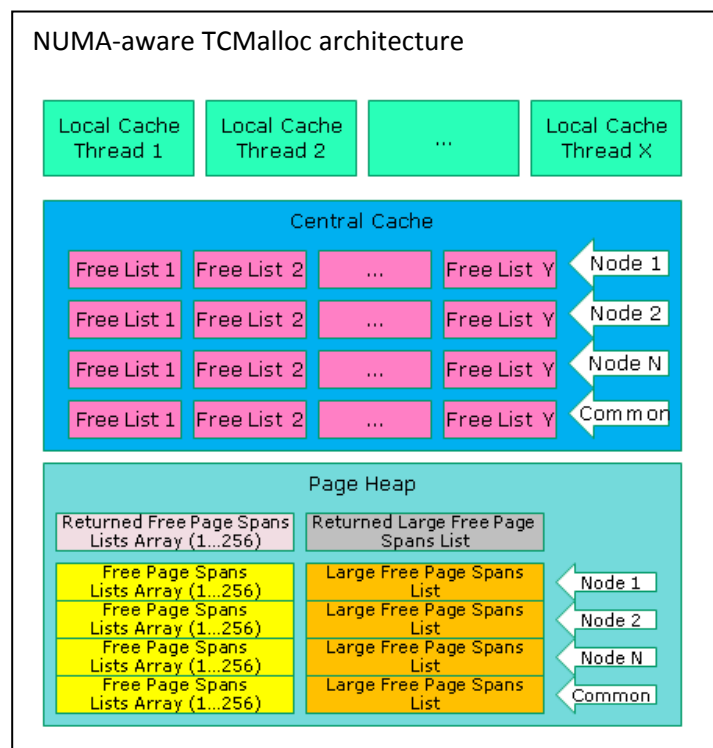
In order to keep the code compatible with Linux and Windows XP and Windows Vista, a decision was made to not rely on advanced NUMA specific APIs since they are not available in Windows XP. This meant that there's no mechanism to allocate memory from a specific NUMA node, or even to let the OS know the preferred node.

It was decided to use the following approach:

- Thread local cache only contains memory blocks used by that thread - TCMalloc assumes that the OS will initially allocate memory local to that thread (per "first-touch" policy)
- The free list array in the central cache is duplicated. There is a separate free list array for each NUMA node in the system plus one extra called Common node (more on this later).
- The Normal Free Page Spans List Array in the Page Heap will be duplicated as well with one for each NUMA node and one extra for the Common node.
 - Note that only the Returned Free Page Spans List Array is not duplicated. This is because it contains virtual memory blocks for which the underlying physical memory has been released back to the operating system; therefore, they are not specific to any NUMA node.

The Common node list (in Central Cache and Page Heap) stores the memory blocks that TCMalloc is no longer certain of the NUMA node mapping.

TCMalloc uses the following criteria to decide what NUMA node a memory block belongs to:



- Initially a memory block is assumed to be allocated on the local NUMA node (per “first-touch” policy). Further, it assumes that the thread which allocated a memory block is the thread that later uses it.
- TCMalloc keeps track of what NUMA node each memory block belongs to. This information is stored in the page span structure in the page heap.
- When a thread frees a memory object, TCMalloc checks if that thread is executing on the same NUMA node that the memory block belongs to
 - If the NUMA node matches, the block is assumed to still belong to that NUMA node
 - If the NUMA node does not match, the block is moved to the Common node (TCMalloc no longer knows which NUMA node the memory block belongs to)
 - Since the NUMA node is recorded in the page span structure, changing the value to Common node automatically changes it for all memory objects allocated from the page span

When a thread tries to allocate a small memory object, TCMalloc first tries to complete the request using memory from the thread’s Local Cache. That memory is assumed to be on the local NUMA node.

If the thread’s Local Cache does not have a free memory object of the specified size, TCMalloc tries to find an appropriate size object in the Free List in the Central Cache. It will only try the list from the local NUMA node.

If the Central Cache does not have a free memory object of the specified size and NUMA locality, it will try to find a free page span list in the Page Heap. Again, TCMalloc will only look for a page span list from the local NUMA node.

If the Page Heap does not have a free page span list for specified NUMA node and the total physical memory usage has not exceeded a specified threshold, TCMalloc attempts to allocate more memory from the operating system. It then designates the newly allocated memory as belonging to the local NUMA node, uses it to populate the Free List in Central Cache, and replenishes the Local Cache for the calling thread.

If the Page Heap does not have a free page span list for the specified NUMA node and the total physical memory usage has exceeded the specified threshold, TCMalloc does not attempt to allocate more memory right away. Instead, it goes back to Central Cache and tries to allocate a right size object from a Free List in the Common node or another NUMA node. If that fails, it tries to allocate a free page span from the Page Heap, again using the common node or

another NUMA node. Finally, if it's still unable to find free memory, it attempts to allocate more memory using an OS API.

For large memory allocations the process is similar except that TCMalloc bypasses the local cache and central cache and tries to find an appropriate free memory block in the page heap.

When a thread calls to release a memory object, TCMalloc checks if the block has been allocated by a thread running on the same NUMA node. If not, the underlying page is marked as belonging to the Common node. If necessary, TCMalloc breaks a larger page span into multiple smaller spans so that each small span can be assigned to a different NUMA node. TCMalloc also aggressively tries to reclaim memory from the Common node. As soon as the page span can be reclaimed, the underlying physical memory is released back to the operating system and the page span is moved to the returned free page span list.

Issues

The above approach has a number of limitations and complex problems, including:

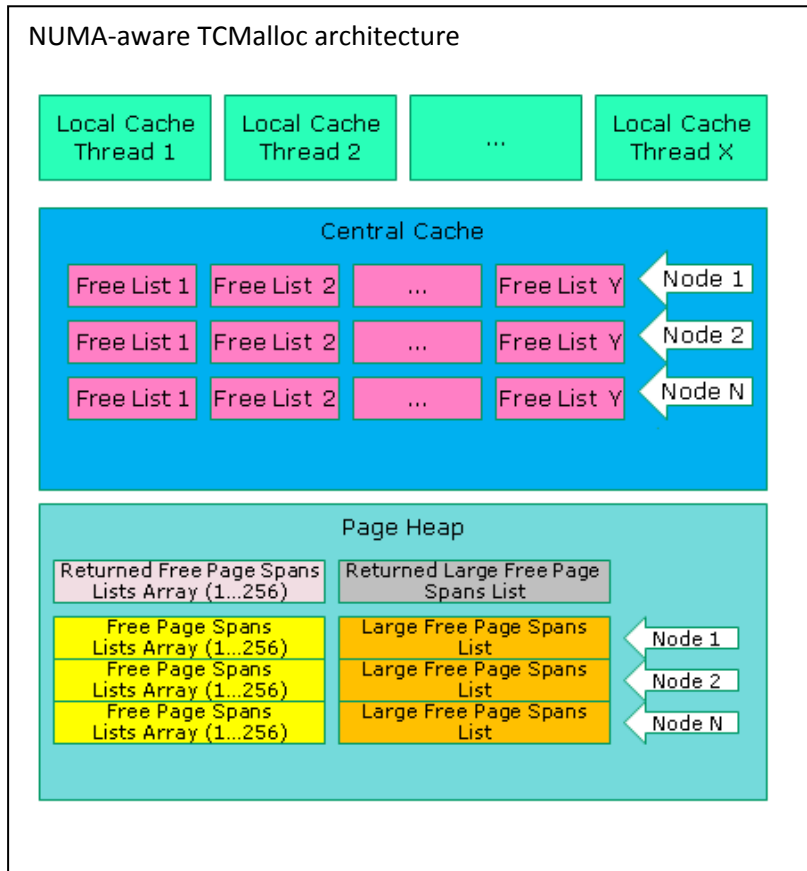
- The basic assumption that a memory block will be used by the same thread which allocated that memory block may be invalid. It is very easy to envision a scenario where a thread allocates a memory object to be used exclusively by another thread. In fact, this is a very common scenario in many applications and, unfortunately, the heap memory manager cannot predict this. Making this assumption can also interfere with an operating system that uses the "first-touch" policy attempting to mitigate this problem.
- TCMalloc should simply check what NUMA node a given memory block is mapped to instead of making any assumptions (which may not always be correct). All operating systems keep track of that information; however, neither Windows nor Linux provide an efficient API to query this information.
- The modified TCMalloc checks the current physical memory usage to determine if it should attempt to allocate more memory or reuse memory from wrong NUMA node. This requires an API that provides the amount of free physical memory that is available on each NUMA node. The [libnuma](#) on Linux provides such functionality (`numa_node_size64`), but the current implementation is very slow. It reads and parses a text file in `/sys/devices/system/node/node<node number>/meminfo`. This is in fact too slow to be called from within TCMalloc. A faster API would be highly desirable.
- A design problem in [libnuma](#) severely limits its usefulness to TCMalloc. TCMalloc needs [libnuma](#) to determine the number of NUMA nodes in the system, map CPUs to NUMA nodes, and to determine physical memory available in each NUMA node. Originally [libnuma](#) `numa_max_node` function (which determined the number of available NUMA nodes) called (indirectly) `malloc`. Since TCMalloc also calls the [libnuma](#) function `numa_max_node` to determine the number of available NUMA nodes, a recursive call to

malloc occurs because libnuma uses CRT functions to open a file. To work around this problem, libnuma has been modified to use POSIX calls instead of CRT to perform the same task.

Second attempt – use NUMA API

Given the problems encountered with the first attempt, it was decided to drop the requirement of maintaining compatibility with Windows XP. This allowed the design to take advantage of an additional API that allows TCMalloc to specify the NUMA node for allocated memory. A similar API is available in both Linux ([libnuma](#) function [mbind](#)) and Windows Vista (function [VirtualAllocExNuma](#)).

The overall architecture of the NUMA-aware TCMalloc remained similar to that in the first attempt, except that the need for the common node was eliminated.



If the Page Heap does not have sufficient memory during memory allocation and TCMalloc allocates more memory from the operating system, the memory will bind to the NUMA node on which the current thread is executing. This guarantees that the underlying physical memory always comes from the specified NUMA node (if available). If the physical memory is released and is later reallocated by the OS, it will still come from the specified NUMA node.

This greatly simplifies NUMA-aware TCMalloc. It no longer needs the Common node since it can always reliably assume the locality of any memory block.

Issues

In spite of the simplifications, the second approach also has a number of issues:

- TCMalloc still needs to check the current physical memory usage to determine if it should attempt to allocate more memory or reuse memory from another NUMA node. Since the Linux API for that task is slow, the modified TCMalloc keeps track of how much memory it already allocated on each node. This is very unreliable since it does not take into account memory usage by other processes.
- The code still tries to release unused physical memory in Page Heap back to the operating system. It only releases the physical memory and keeps the virtual memory allocated. On Linux this is done using the [madvise](#) function call with the MADV_DONTNEED flag. Since Windows does not have an equivalent API, similar functionality is available using the [VirtualFreeEx](#) with the MEM_DECOMMIT flag. When the memory is later reclaimed on Linux no additional work is required, but on Windows TCMalloc must call [VirtualAllocExNuma](#) to again commit the physical memory. There is a significant difference between the two calls. On Linux, if an application touches the virtual memory region for which the physical memory has been released (using the [madvise](#) call), there will be no side effects other than the possible delay required to commit a new physical memory page. On Windows, if an application tries to access a memory region that has been de-committed (using the [VirtualFreeEx](#) call) it will result in a system exception. Although an application should never access a memory block that has been de-committed, it may happen. Even though that would be a bug, on Linux it might go unnoticed, whereas on Windows it will result in an application crash (unless it has an exception handler to catch it).

The second approach was chosen because it greatly simplifies the code changes and results in almost identical performance. In the future however, it might be worth revisiting the first approach, especially if the OS provides an efficient API to determine the free physical memory on each NUMA node and/or to map a virtual memory address to the underlying NUMA node.

Performance analysis

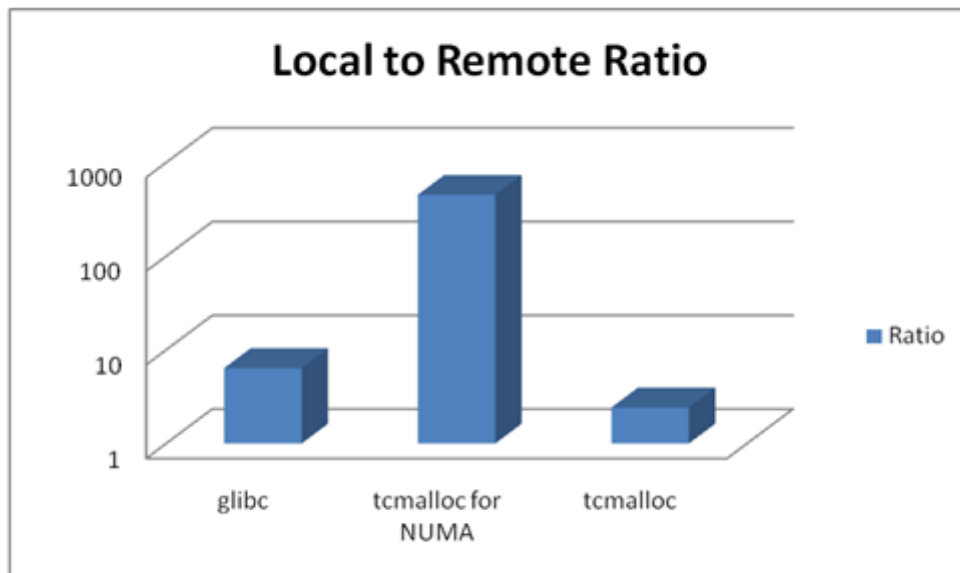
To test the NUMA-aware TCMalloc, a new synthetic benchmark has been implemented. The goal of the benchmark was to simulate the ideal conditions assumed by the NUMA-aware TCMalloc. The benchmark starts multiple worker threads. Each worker thread repeatedly allocates a memory block, performs very memory intensive operations, and then releases the memory block. The maximum allocated memory was kept under the total physical memory available.

First the benchmark code that actually performed the memory accesses was commented out (so it only allocated and immediately released the memory blocks). When the benchmark was built against the original TCMalloc and NUMA-aware TCMalloc, there was no noticeable performance difference.

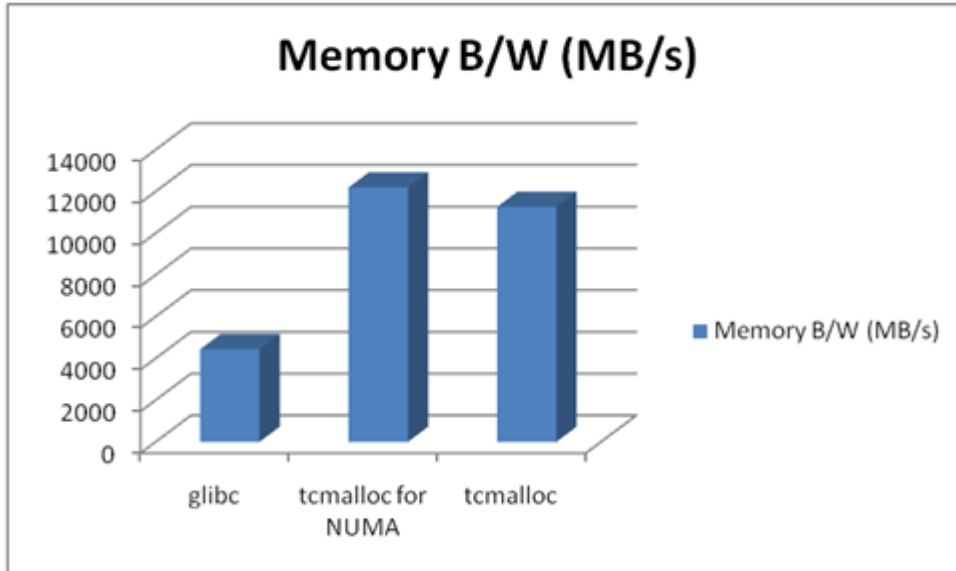
Next the benchmark code that performed the memory accesses was placed back in. This time the NUMA-aware TCMalloc had almost 40% performance improvement on a quad-processor K8 machine. The results imply that the performance difference was caused by better memory management (local vs. remote, reduced HyperTransport™ bus overhead, etc.).

Note that this was under ideal conditions, best-case scenario. In real life, for most applications the performance difference is likely to be much lower.

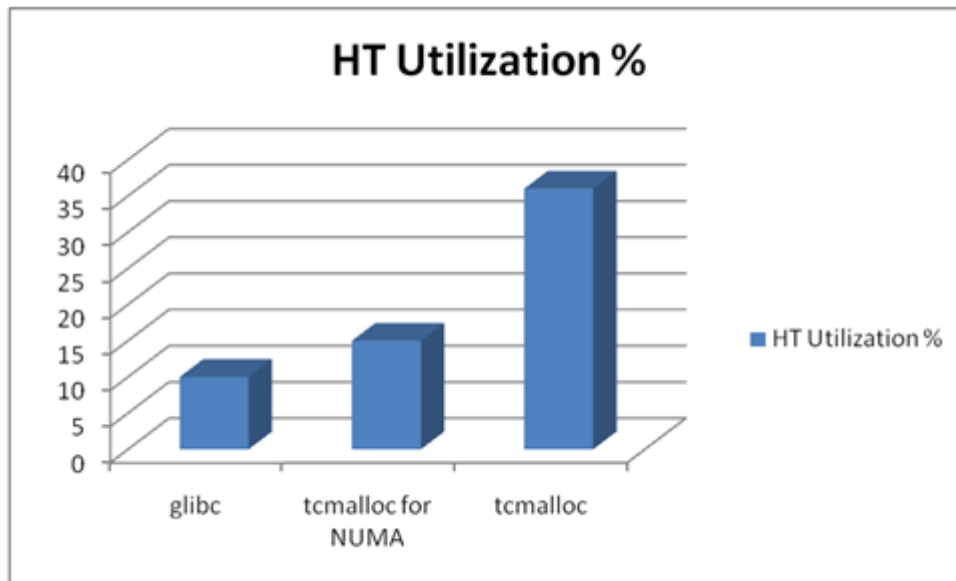
Additional measurements were done to determine the local to remote memory access ratio, memory bandwidth and HyperTransport bus utilization. The following charts display the results on a dual-processor K8 machine.



This clearly shows a significant advantage for NUMA-aware TCMalloc. Most of the memory accesses in the benchmark were local, as opposed to glibc or tcmalloc. It is suspected that glibc turned out better in this chart (compared to the original TCMalloc) simply because of the advanced memory caching techniques in TCMalloc.



Memory bandwidth also shows NUMA-aware TCMalloc as a clear winner, although the actual performance advantage is not as significant.



NUMA-aware TCMalloc also shows a much better HyperTransport bus utilization than the original TCMalloc on a dual-processor system. The glibc implementation has an even lower utilization, probably because of significantly less overall performance.

The benchmarks were compiled with GCC version 4.2.1. The glibc library version used in the tests was 2.6.1. The Linux distribution used was Open SUSE 10.3 with kernel 2.6.25-rc8.

Suitable real world problems?

The author has tried to find a good real-world problem to test the NUMA-aware TCMalloc. Here are some of the applications that have been tried:

- Dell DVD Store (DS2) benchmark on MySQL
- Dell DVD Store (DS2) benchmark on Progress SQL
- POV-Ray
- Some benchmarks for Apache HTTP server
- Several other proprietary applications/benchmarks

Unfortunately, finding a suitable benchmark turned out to be a very difficult task. To really show the benefits of NUMA optimizations one needs a very memory intensive application. Tuning MySQL or Progress SQL to achieve a high (95+%) CPU utilization without dedicated hardware, fiber optics network adapters, etc., turned out to be extremely difficult. POV-Ray uses its own memory management techniques, which prevented the author from easily plugging in TCMalloc. Other benchmarks were either already optimized for NUMA, or could not achieve the required CPU utilization.

Recommendations for future operating systems

During the development of the NUMA aware TCMalloc, it became clear that modern operating systems lack key APIs required to effectively manage memory in NUMA systems. These APIs include:

- A method to query the NUMA node to virtual memory mapping.
- A way to efficiently query the amount of free physical memory in each NUMA node.
- Ability to determine what NUMA node the current thread is running on.
- A better mechanism to detect and control if the operating system moves a thread execution from one process to another for load balancing.
- A simplified mechanism to substitute a standard Heap Memory manager in the CRT with a custom one as is available in GLIBC, but not in other CRT implementations.

Future ideas for NUMA aware TCMalloc

Here are several ideas that may be worth implementing in the future:

- Once Linux or Windows provides an efficient API to determine what NUMA node a given memory block is mapped to, it might be worthwhile to modify TCMalloc to use that API. This would eliminate the need for TCMalloc to make any assumptions about the NUMA node. In particular, it might help in a situation where the thread that allocated a memory block was not the first thread to use that memory block.
- A reliable and fast method for determining the free physical memory on a specified NUMA node would eliminate the need for TCMalloc to estimate the memory usage.

- A more sophisticated decision mechanism for determining whether to request more memory from the operating system or reuse already allocated memory from another NUMA node would be very helpful. The current mechanism is very simplistic and relies on a single threshold value.
- A more sophisticated decision mechanism that takes into account NUMA topology would allow TCMalloc to allocate memory on local NUMA nodes and if that is not possible, it could try the next closest NUMA node (based on the topology of the system), and so on. Currently TCMalloc tries to allocate memory on a local node, or any other NUMA node.
- Currently if the thread local cache does not have enough free memory, TCMalloc will look into Central Cache. The Central Cache is protected by a single global lock. Multiple global locks, one per NUMA node, could possibly reduce the lock contention.

Source Code

Prerequisites

There is one dependency for NUMA-aware TCMalloc on Linux: [libnuma](#). As has been mentioned above, libnuma has an implementation issue where the [numa_max_node](#) function calls malloc (indirectly via a CRT file I/O routine). Therefore, TCMalloc cannot call this function from its own malloc function implementation. The author has modified libnuma to work around this problem by using only POSIX calls to do the file IO. Note, however, that the second implementation works around this problem by not calling this function from malloc eliminating the need for the libnuma patch.

- [Libnuma patch](#)

The numa.zip archive contains the only file (libnuma.c) that was modified in the library. It should be applied on top of numactl-1.0.2.

NUMA-aware TCMalloc

The complete source code of NUMA-aware TCMalloc is provided on an as-is basis. It is based on the google-perftools-0.97 code base and includes a diff against version 0.97.

- [Source code](#)
- [Diff](#)

Benchmark code

The source code of the simple synthetic benchmark that was used to test the performance of the NUMA-aware TCMalloc is also provided.

- [Benchmark code](#)

It was tested on a 4 processor system (each processor was a dual core Opteron 870, 2 GHz) with 8 GB RAM (2 GB per CPU socket).

The benchmark starts 8 worker threads (one thread for each logical CPU core in the system) as follows:

- Each worker thread runs in a loop 5000 times
 - Allocates a random size (between 64 bytes and 64 MB) block of memory. (size = $2^{(6 + \text{rand}() \% 19)}$)
 - Accesses the memory (multiple reads and writes):
 - Repeat 10 times:
 - Read and write every 64th byte of the memory block
 - Releases the memory block

The benchmark measures the time required for all worker threads to complete.

The benchmark was designed to simulate an optimal scenario where there is sufficient memory available from each NUMA node and simulates an application that uses the memory very intensively.

The following results were observed:

- Code using original TCMalloc runs approximately 8% slower than code using GLIBC heap manager
- Code using NUMA aware TCMalloc runs approximately 32% faster than code using GLIBC heap manager
- Code using NUMA aware TCMalloc runs approximately 37% faster than code using original TCMalloc heap manager

© 2009 Advanced Micro Devices, Inc. All rights reserved.

AMD, the AMD logo, ATI, the ATI logo, AMD Opteron, and combinations thereof are trademarks of Advanced Micro Devices, Inc. HyperTransport is a licensed trademark of the HyperTransport Technology Consortium. Microsoft, Windows, and Windows Vista are registered trademarks of Microsoft Corporation in the United States and/or other jurisdictions. Linux is a registered trademark of Linux Torvalds. All other names used in this paper are for informational purposes only and may be the trademarks of their respective companies.