Performance Guidelines for AMD Athlon™ 64 and AMD Opteron™ ccNUMA Multiprocessor Systems

Application Note

Publication # **40555** Revision: 3.00
Issue Date: **June 2006**

Trademarks

Contents

List of Figures

Revision History

Date	Revision	Description
June 2006	3.00	Initial release.

Chapter 1 Introduction

The AMD Athlon™ 64 and AMD Opteron™ family of single-core and dual-core multiprocessor systems are based on the cache coherent Non-Uniform Memory Access (ccNUMA) architecture. In this architecture, each processor has access to its own low-latency, local memory (through the processor's on-die local memory controller), as well as to higher latency remote memory through the on-die memory controllers of the other processors in the multiprocessor environment. At the same time, the ccNUMA architecture is designed to maintain the cache coherence of the entire shared memory space. The high-performance coherent HyperTransport™ technology interconnects between processors in the multiprocessor system permit remote memory access and cache coherence.

In traditional symmetric multiprocessing (SMP) systems, the various processors share a single memory controller. This single memory connection can become a performance bottleneck when all processors access memory at once. At the same time, the SMP architecture does not scale well into larger systems with a greater number of processors. The AMD ccNUMA architecture is designed to overcome these inherent SMP performance bottlenecks. It is a mature architecture that is designed to extract greater performance potential from multiprocessor systems.

As developers deploy more demanding workloads on these multiprocessor systems, common performance questions arise: Where should threads or processes be scheduled (thread or process placement)? Where should memory be allocated (memory placement)? The underlying operating system (OS), tuned for AMD Athlon 64 and AMD Opteron multiprocessor ccNUMA systems, makes these performance decisions transparent and easy.

Advanced developers, however, should be aware of the more advanced tools and techniques available for performance tuning. In addition to recommending mechanisms provided by the OS for explicit thread (or process) and memory placement, this application note explores advanced techniques such as node interleaving of memory to boost performance. This document also delves into the characterization of an AMD ccNUMA multiprocessor system, providing advanced developers with an understanding of the fundamentals necessary to enhance the performance of synthetic and real applications and to develop advanced tools.

In general, applications can be memory latency sensitive or memory bandwidth sensitive; both classes are important for performance tuning. In a multiprocessor system, in addition to memory latency and memory bandwidth, other factors influence performance:

- the latency of remote memory access (*hop latency*)
- the latency of maintaining cache coherence (*probe latency*)
- the bandwidth of the HyperTransport interconnect links
- the lengths of various buffer queues in the system

The empirical analysis presented in this document is based upon data provided by running a multi-threaded synthetic test. While this test is neither a pure memory latency test nor a pure memory

bandwidth test, it exercises both of these modes of operation. The test serves as a latency sensitive test case when the test threads perform read-only operations and as a bandwidth sensitive test when the test threads carry out write-only operations. The discussion below explores the performance results of this test, with an emphasis on behavior exhibited when the test imposes high bandwidth demands on the low level resources of the system.

Additionally, the tests are run in *undersubscribed, highly subscribed*, and *fully subscribed* modes. In undersubscribed mode, there are significantly fewer threads than the number of processors. In highly subscribed mode, the number of threads approaches the number of processors. In the fully subscribed mode, the number of threads is equal to the number of processors. Testing these conditions provides an understanding of the impact of thread subscription on performance.

Based on the data and the analysis gathered from this synthetic test-bench, this application note presents recommendations to software developers who are working on applications, compiler tool chains, virtual machines and operating systems. Finally, the test results should also dispel some common myths concerning identical performance results obtained when comparing workloads that are symmetrical in all respects except for the thread and memory placement used.

1.1 Related Documents

The following web links are referenced in the text and provide valuable resource and background information:

[1] http://www.hotchips.org/archives/hc14/3_Tue/28_AMD_Hammer_MP_HC_v8.pdf

[2] http://www.kernel.org/pub/linux/kernel/people/mbligh/presentations/OLS2004-numa_paper.pdf

[3] http://www.amd64.org/lists/discuss/msg03314.html

[4] http://www.pgroup.com/doc/pgiug.pdf

[5] http://www.novell.com/collateral/4621437/4621437.pdf

[6] http://opensolaris.org/os/community/performance/mpo_overview.pdf

[7] http://www.opensolaris.org/os/community/performance/numa/observability/

[8] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/multiple_processors.asp

[9] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/virtualalloc.asp

[10] http://msdn2.microsoft.com/en-us/library/ms186255(SQL.90).aspx

[11] http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/529588d3-71bc-45ea-a84b-267914674709.mspx

[12] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dngenlib/html/msdn_heapmm.asp

[13] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/low_fragmentation_heap.asp

[14] http://msdn2.microsoft.com/en-us/library/tt15eb9t.aspx

[15] https://www.pathscale.com/docs/UserGuide.pdf

[16] http://docs.sun.com/source/819-3688/parallel.html

Chapter 2 Experimental Setup

This chapter presents a description of the experimental environment within which the following performance study was carried out. This section describes the hardware configuration and the software test framework used.

2.1 System Used

All experiments and analysis discussed in this application note were performed on a Quartet system having four 2.2 GHz E1 Dual-Core AMD Opteron™ processors running the Linux® 2.6.12-rc1-mm1 kernel (ccNUMA-aware kernel).

While Quartet is an internal development non-commercial platform, the way the processors are connected on the Quartet is a common way of connecting and routing the processors on other supported 4P AMD platforms. We anticipate that these results should hold on other systems that are connected in a similar manner and we expect the recommendations to carry forward on the current generation Opteron systems. We also expect that the results will hold on other Linux kernels and even other operating systems for reasons explained later.

Each processor had 2x1GB DDR400 CL2.5 PC3200 (Samsung K4H510838B-TCCC) DRAM memory. To rule out any interference from the Xserver and the network, all tests were performed at runlevel 3 with the network disconnected.

At a high level, in a Quartet, the four dual-core processors are connected with coherent HyperTransport™ links. Each processor has one bidirectional HyperTransport link that is dedicated to I/O and two bidirectional coherent HyperTransport links that are used to connect to two other dual-core processors. This enables a direct connection for a given dual-core processor to all other dual-core processors but one in a 4-way configuration. The throughput of each bidirectional HyperTransport link is 4 GB/s in each direction. Each node has its own on-chip memory controller and is connected to its own memory.

As shown in Figure 1 on page 14, the processors (also called *nodes*) are numbered N0, N1, N3 and N2 clockwise from the top left. Each Node has two cores—labeled C0 and C1 respectively [1].

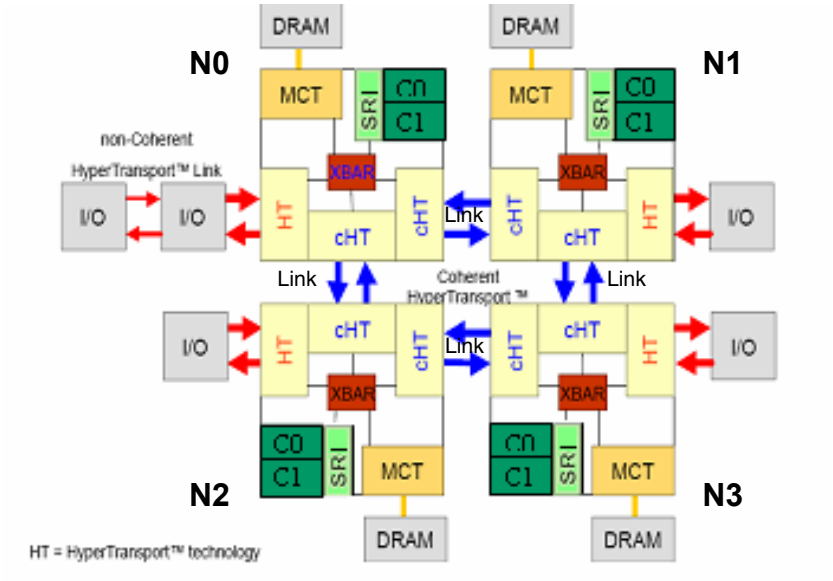

Figure 1. Quartet Topology

The term *hop* is commonly used to describe access distances on NUMA systems. When a thread accesses memory on the same node as that on which it is running, it is a 0-hop access or *local* access. If a thread is running on one node but accessing memory that is resident on a different node, the access is a *remote* access. If the node on which the thread is running and the node on which the memory resides are directly connected to each other, the memory access is a 1-hop access. If they are indirectly connected to each other (i.e., there is no direct coherent HyperTransport link) in the 4P configuration shown in Figure 1, the memory access is a 2-hop access. For example, if a thread running on Node 0 (N0) accesses memory resident on Node 3 (N3), the memory access is a 2-hop access.

Figure 2 on page 15 views the resources of each node from a lower level perspective. Each (dual-core) processor has two cores. The two cores talk to a system request interface (SRI), which in turn talks to a crossbar (XBar). The crossbar is connected to the local memory controller (MCT) on one end and to the various HyperTransport links on the other end. The SRI, XBar and MCT are collectively called the Northbridge on the node. The MCT is connected to the physical memory (DRAM) for that node.

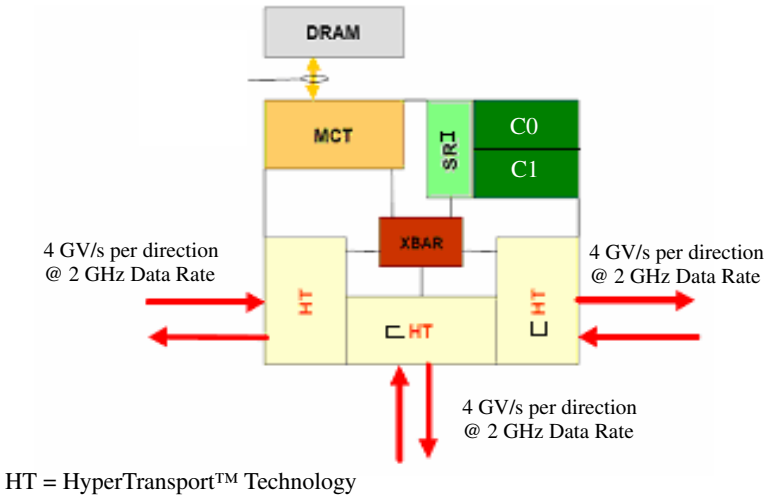

Figure 2. Internal Resources Associated with a Quartet Node

From the perspective of the MCT, a memory request may come from either the local core or from another core over a coherent HyperTransport link. The former request is a *local* request, while the latter is a *remote* request. In the former case, the request could be routed from the local core to the SRI, then to the XBar and then to the MCT. In the later case, the request is routed from the remote core over the coherent HyperTransport link to the XBar and from there to the MCT.

The MCT, the SRI and the XBar on each node all have internal buffers that are used to queue transaction packets for transmission. For additional details on the Northbridge buffer queues, refer to Section A.1 on page 39.

From a system perspective, the developer can think of the system as having three key resources that affect throughput: memory bandwidth, HyperTransport bandwidth and buffer queue capacity.

2.2 Synthetic Test

The test used is a simple synthetic workload consisting of two threads with each thread accessing an array that is not shared with the other thread. The time taken by each thread to access this array is measured.

Each thread does a series of read-only or write-only accesses to successive elements of the array using a cache line stride (64 bytes). The test iterates through all permutations of read-read, read-write, write-read, and write-write for the access patterns of the two threads. Each array is sized at 64MB—significantly larger than the cache size.

This synthetic test is neither a pure memory latency test nor a pure memory bandwidth test; rather it places varying throughput and capacity demands on the resources of the system described in the previous section. This provides an understanding of how the system behaves when any of the

resources approach saturation. The test has two modes: *read-only* and *write-only*. When the test threads are read-only, the throughput does not stress the capacity of the system resources and, thus, the test is more sensitive to latency. However, when the threads are write-only, there is a heavy throughput load on the system. This is described in detail in later sections of this document.

Each thread is successively placed on all possible cores in the system. The data (array) accessed by each thread is also successively placed on all possible nodes in the system. Several Linux application programming interfaces (APIs) are used to explicitly pin a thread to a specified core and data to a specified node, thus allowing full control over thread and memory placement. (For additional details on the Linux API refer to section A.1 on page 39.) Once a thread or data is pinned to a core or node, it remains resident there for its entire lifetime. Thus the test runs through all permutations of thread and memory placement possible for the two threads. Since the test does not rely on the OS for thread and memory placement, the results obtained from the test are independent of the low level decisions made by the OS and are thus OS agnostic.

First, the two thread experiments are run on an idle system, thereby generating a truth table of 4096 timing entries for the two threads. The results are then mined to evaluate interesting scenarios of thread and memory placement. Several of these scenarios are presented in various graphs in this document.

Next, the experiments are enhanced by adding a variable load of background threads. The behavior of the two test (or foreground) threads is studied under the impact of these variable load background threads.

Each of the background threads reads a local 64-MB array. The rate at which each background thread accesses memory can be adjusted from *low* to *medium* to *high* to *very high* to control the background load. Table 1 defines these rate qualifiers.

Table 1. Data Access Rate Qualifiers

Data Access Rate Qualifier	Memory Bandwidth Demanded by a Background Thread on an Idle System
Low	0.5 GB/s
Medium	1 GB/s
High	2 GB/s
Very High	4 GB/s

The number of background threads is also varied as needed to make an increasing number of cores and nodes on the system busy—in other words, to increase the subscription. Full subscription means that every core in the system is busy running a thread. High subscription means that while several cores are busy, there are still some cores left free in the system.

The data-mining suggested several basic recommendations for performance enhancement on these systems. Also revealed were some interesting cases of asymmetry that allowed the low level

characterization of the resource behavior in the system. These recommendations, coupled with these interesting cases, provide an understanding of the low-level behavior of the system, which is crucial to the analysis of larger real-world workloads.

2.3 Reading and Interpreting Test Graphs

Figure 3 below shows one of the graphs that will be discussed in detail later.

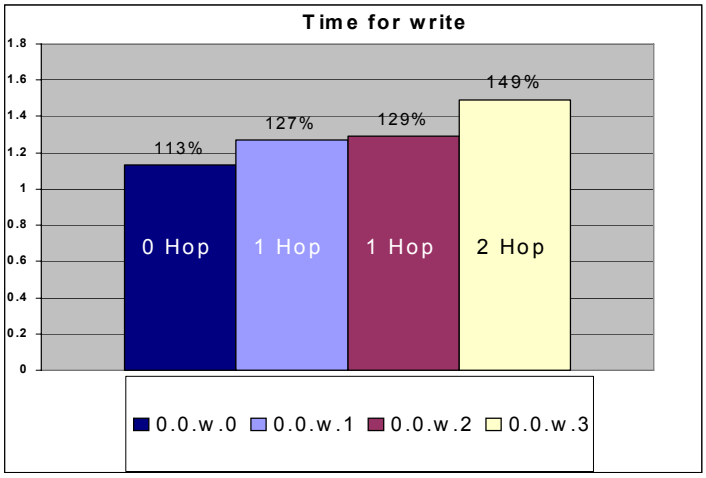

Figure 3. Write-Only Thread Running on Node 0, Accessing Data from 0, 1 and 2 Hops Away on an Idle System

2.3.1 X-Axis Display

The X-axis displays the various thread and memory placement combinations to be compared. Each case shows the following information for the thread, in the order listed:

- The node and core on which the thread is run.
- Whether accesses are read-only or write-only.
- The node on whose physical memory the data accessed by the thread resides.

In Figure 3 above, the four bars correspond to the following thread combinations:

0.0.w.0—Thread running on node 0/core0 does write-only accesses to memory resident on node 0.

0.0.w.1—Thread running on node 0/core0 does write-only accesses to memory resident on node 1.

0.0.w.2—Thread running on node 0/core0 does write-only accesses to memory resident on node 2.

0.0.w.3—Thread running on node 0/core0 does write-only accesses to memory resident on node 3.

2.3.2 Labels Used

Each of the bars on the graph is labeled with the hop information for the thread.

2.3.3 Y-Axis Display

For the one-thread test cases on the idle system, the graphs show the time taken by a single thread, normalized to the time taken by the fastest single-thread case—in this case the time it takes a read-only thread to do local accesses on an idle system.

In Figure 3 on page 17, the time taken by the 0.0.w.0 case is normalized to the time taken by 0.0.r.0 case. The reason for the different times recorded for these two cases is explained later.

For the two-thread test cases on an idle system, the graphs show the total time taken by both threads, normalized to the time taken by the fastest two-thread case—in this case the time it takes two read-only threads running on different nodes to do local accesses on an idle system.

These graphs are further enhanced by rerunning the two-thread cases with some background threads.

For the two-thread test cases with a background load, the graphs show the total time taken by the two test (foreground) threads normalized as before. The time taken by the background threads is not measured. Only the effect of background threads on the performance of test threads is evaluated by measuring the time taken by the test threads. The load imposed by the background threads is varied from low to very high and graphs are plotted for the individual load scenarios.

Chapter 3 Analysis and Recommendations

This section lays out recommendations to developers. Several of these recommendations are accompanied by empirical results collected from test cases with analysis, as applicable.

In addition to making recommendations for performance improvement, this section clarifies some of the common perceptions developers have about performance on AMD ccNUMA systems and, at the same time, reveals the impact of low level system resources on performance. The extent of the impact of these resources on the performance of any given application depends on the nature of the application. The goal is to help developers think like the machine when interpreting "counter intuitive" behavior while performance tuning.

While all analysis and recommendations are made with reference to the context of threads, they can also be applied to processes.

3.1 Scheduling Threads

Scheduling multiple threads across nodes and cores of a system is complicated by a number of factors:

- Whether the system is idle.
- Whether multiple threads access independent data.
- Whether multiple threads access shared data.

3.1.1 Multiple Threads-Independent Data

When scheduling multiple threads which access independent data on an idle system, it is preferable first to schedule the threads to an idle core of each node until all nodes are exhausted and then schedule the other idle core of each node. In other words, schedule using node major order first, followed by core major order. This is the suggested policy for a ccNUMA aware operating system on an AMD dual-core multiprocessor system.

For example, when scheduling threads, which access independent data, on the dual-core Quartet, scheduling the threads in the following order is recommended:

- Core 0 on node 0, node 1, node 2 and node 3 in any order
- Core 1 on node 0, node 1, node 2 and node 3 in any order

The two cores on each node of the dual-core AMD Opteron™ processor share the Northbridge resources, which include the memory controller and the physical memory that is connected to that node. The main motivation for this recommendation is to avoid overloading the resources on a single node, while leaving the resources on the rest of the system unused—in other words load balancing.

3.1.2 Multiple Threads-Shared Data

When scheduling multiple threads that share data on an idle system, it is preferable to schedule the threads on both cores of an idle node first, then on both cores of the the next idle node, and so on. In other words, schedule using core major order first followed by node major order.

For example, when scheduling threads that share data on a dual-core Quartet system, AMD recommends using the following order:

- Core 0 and core 1 on node 0 in any order
- Core 0 and core 1 on node 1 in any order
- Core 0 and core 1 on node 2 in any order
- Core 0 and core 1 on node 3 in any order

3.1.3 Scheduling on a Non-Idle System

Scheduling multiple threads for an application optimally on a non-idle system is a more difficult task. It requires that the application make global holistic decisions about machine resources, coordinate itself with other applications already running, and balance decisions between them. In such cases, it is better to rely on the OS to do the appropriate load balancing [2].

In general, most developers will achieve good performance by relying on the ccNUMA-aware OS to make the right scheduling decisions on idle and non-idle systems. For additional details on ccNUMA scheduler support in various operating systems, refer to Section A.6 on page 43.

In addition to the scheduler, several NUMA-aware OSs provide tools and application programming interfaces (APIs) that allow the developer to explicitly set thread placement to a certain core or node. Using these tools or APIs overrides the scheduler and hands over control for thread placement to the developer, who should use the previously mentioned techniques to assure reasonable scheduling.

For additional details on the tools and API libraries supported in various OSs, refer to Section A.7 on page 44.

3.2 Data Locality Considerations

It is best to keep data local to the node from which it is being accessed. Accessing data remotely is slower than accessing data locally. The further the hop distance to the data, the greater the cost of accessing remote memory. For most memory-latency sensitive applications, keeping data local is the single most important recommendation to consider.

As explained in Section 2.1 on page page 13, if a thread is running and accessing data on the same node, it is considered as a local access. If a thread is running on one node but accessing data resident on a different node, it is considered as a remote access. If the node where the thread is running and the node where the data is resident are directly connected to each other, it is considered as a 1 hop access

distance. If they are indirectly connected to each other in a 4P configuration, it is considered as a 2 hop access distance.

The following example—extracted from mining the results of the synthetic test case—substantiates the recommendation to keep data local.

In this test, a single thread ran on node 0 (core 0) on an otherwise idle system and each of the following cases were measured and compared:

- Thread accessed data locally from node 0
- Thread accessed data one hop away from node 1
- Thread accessed data one hop away from node 2
- Thread accessed data two hops away from node 3

As seen in Figure 4 and Figure 5 on page 22, as the hop distance increases, access times increase for both reads and writes.

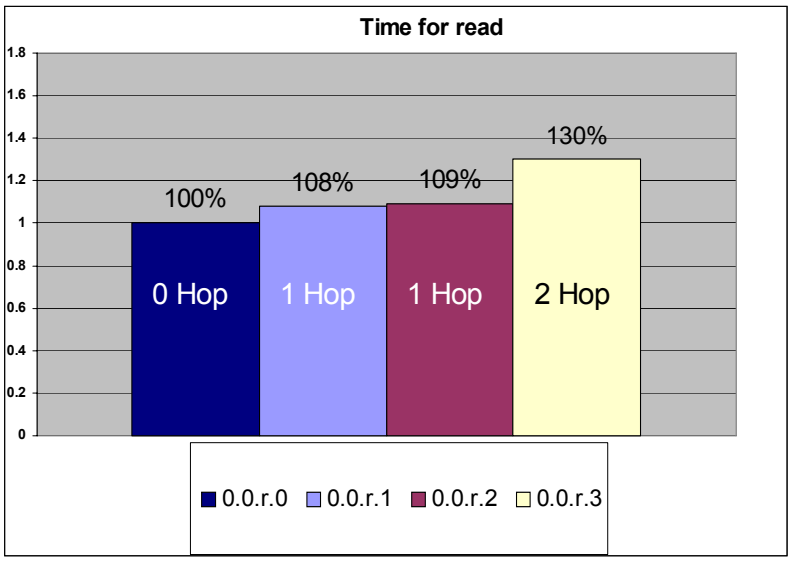

Figure 4. Read-Only Thread Running on Node 0, Accessing Data from 0, 1 and 2 Hops Away on an Idle System

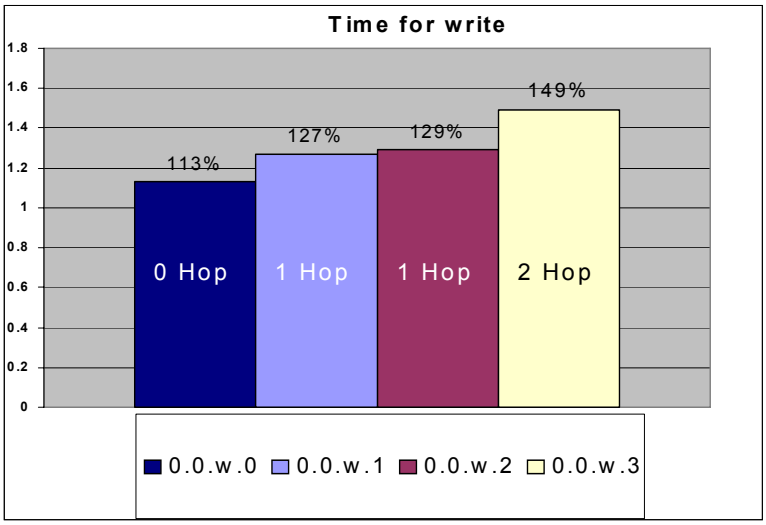

Figure 5. Write-Only Thread Running on Node 0, Accessing Data from 0, 1 and 2 Hops Away on an Idle System

In this test case, a write access is similar to a read access as far as the coherent HyperTransport™ link traffic or the memory traffic generated, except for certain key differences. A write access brings data into the cache much like a read and then modifies it in the cache. However, in this particular synthetic test case, there are several successive write accesses to sequential cache line elements in a 64-MB array. This results in a steady state condition of cache line evictions or write-backs for each write access. This increases the memory and HyperTransport traffic that normally occurs for a write-only thread to almost twice that of a read-only thread. For our test bench, when a thread does local read-only accesses, it generates almost twice the memory bandwidth load of 1.64 GB/s, and when a thread performs local write-only accesses, it generates a memory bandwidth load of 2.98 GB/s. Not only do writes take longer than reads for any given hop distance, but they slow down more quickly with hop distance as a result.

3.2.1 Keeping Data Local by Virtue of first Touch

In order to keep data local, it is recommended that the following principles be observed.

As long as a thread initializes the data it needs (writes to it for the first time) and does not rely on any other thread to perform the initialization, a ccNUMA-aware OS keeps data local on the node where the thread runs. This policy of keeping data local by writing to it for the first time is known as the *local allocation* policy by virtue of *first touch*. This is the default policy used by a ccNUMA-aware OS.

A ccNUMA-aware OS ensures local allocation by taking a page fault at the time of the first touch to data. When the page fault occurs the OS maps the virtual pages associated with the data to zeroed out physical pages. Now the data is resident on the node where the first touch occurred and any subsequent accesses to the data will have to be serviced from that node.

A ccNUMA-aware OS keeps data local on the node where first-touch occurs as long as there is enough physical memory available on that node. If enough physical memory is not available on the node, then various advanced techniques are used to determine where to place the data, depending on the OS.

Data once placed on a node due to first touch normally resides on that node for its lifetime. However, the OS scheduler can migrate the thread that first touched the data from one core to another core—even to a core on a different node. This can be done by the OS for the purpose of load balancing [3].

This migration has the effect of moving the thread farther from its data. Some schedulers try to bring the thread back to a core on a node where the data is in local memory, but this is never guaranteed. Furthermore, the thread could first touch more data on the node to which it was moved before it is moved back. This is a difficult problem for the OS to resolve, since it has no prior information as to how long the thread will run and, hence, whether migrating it back is desirable or not.

If an application demonstrates that threads are being moved away from their associated memory by the scheduler, it is typically useful to explicitly set thread placement. By explicitly pinning a thread to a node, the application can tell the OS to keep the thread on that node and, thus, keep data accessed by the thread local to it by the virtue of first touch.

The performance improvement obtained by explicit thread placement may vary depending on whether the application is multithreaded, whether it needs more memory than available on a node, whether threads are being moved away from their data, etc.

In some cases, where threads are scheduled from the outset on a core that is remote from their data, it might be useful to explicitly control the data placement. This is discussed in detail in the Section 3.2.2.

The previously discussed tools and APIs for explicitly controlling thread placement can also be used for explicitly controlling data placement. For additional details on thread and memory placement tools and API in various OS, refer to Section A.7 on page 44.

3.2.2 Data Placement Techniques to Alleviate Unnecessary Data Sharing Between Nodes Due to First Touch

When data is shared between threads running on different nodes, the default policy of local allocation by first touch used by the OS can become non-optimal.

For example, a multithreaded application may have a startup thread that sets up the environment, allocates and initializes a data structure and forks off worker threads. As per the default local allocation policy, the data structure is placed on physical memory of the node where the start up thread did the first touch. The forked worker threads are spread around by the scheduler to be balanced across all nodes and their cores. A worker thread starts accessing the data structure remotely from the memory on the node where the first touch occurred. This could lead to significant memory and HyperTransport traffic in the system. This makes the node where the data resides the bottleneck. This situation is especially bad for performance if the startup thread only does the initialization and

afterwords no longer needs the data structure and if only one of the worker threads needs the data structure. In other words, the data structure is not truly shared between the worker threads.

It is best in this case to use a data initialization scheme that avoids incorrect data placement due to first touch. This is done by allowing each worker thread to first touch its own data or by explicitly pinning the data associated with each worker thread on the node where the worker thread runs.

Certain OSs provide memory placement tools and APIs that also permit data migration. A worker thread can use these to migrate the data from the node where the start up thread did the first touch to the node where the worker thread needs it. There is a cost associated with the migration and it would be less efficient than using the correct data initialization scheme in the first place.

If it is not possible to modify the application to use a correct data initialization scheme or if data is truly being shared by the various worker threads—as in a database application—then a technique called node interleaving can be used to improve performance. Node interleaving allows for memory to be interleaved across any subset of nodes in the multiprocessor system. When the node interleaving policy is used, it overrides the default local allocation policy used by the OS on first touch.

Let us assume that the data structure shared between the worker threads in this case is of size 16 KB. If the default policy of local allocation is used then the entire 16KB data structure resides on the node where the startup thread does first touch. However, using the policy of node interleaving, the 16-KB data structure can be interleaved on first touch such that the first 4KB ends up on node 0, the next 4KB ends up on node 1, and the next 4KB ends up on node 2 and so on. This assumes that there is enough physical memory available on each node. Thus, instead of having all memory resident on a single node and making that the bottleneck, memory is now spread out across all nodes.

The tools and APIs that support explicit thread and memory placement mentioned in the previous sections can also be used by an application to use the node interleaving policy for its memory. For additional details refer to Section A.8 on page 46.

By default, the granularity of interleaving offered by the tools/APIs is usually set to the size of the virtual page supported by the hardware, which is 4K (when system is configured for normal pages, which is the default) and 2M ((when system is configured for large pages,). Therefore any benefit from node interleaving will only be obtained if the data being accessed is significantly larger than a virtual page size.

If data is being accessed by three or more cores, then it is better to interleave data across the nodes that access the data than to leave it resident on a single node. We anticipate that using this rule of thumb could give a significant performance improvement. However, developers are advised to experiment with their applications to measure any performance change.

A good example of of the use of node interleaving is observed with Spec JBB 2005 using Sun JVM 1.5.0_04-GA. Using node interleaving improved the peak throughput score reported by Spec JBB 2005 by 8%. We observe that, as this benchmark starts with a single thread and then ramps up to eight threads, all threads end up accessing memory resident on a single node by the virtue of first touch.

Spec JBB 2005 was run using the NUMA tools provided by Linux® to measure the performance improvement with node interleaving. The results were obtained on the same internal 4P Quartet system used for the synthetic tests.

3.3 Avoid Cache Line Sharing

In a ccNUMA multiprocessor system, data within a single cache line that is shared between cores, even on the same node, can reduce performance. In certain cases, such as semaphores, this kind of cache-line data sharing cannot be avoided, but it should be minimized where possible.

Data can often be restructured so that such cache-line sharing does not occur. Cache lines on AMD Athlon™ 64 and AMD Opteron™ processors are currently 64 bytes, but a scheme that avoids this problem, regardless of cache-line size, makes for more performance-portable code. For example, a multithreaded application should avoid using statically defined shared arrays and variables that are potentially located in a single cache line and shared between threads.

3.4 Common Hop Myths Debunked

This section addresses several commonly held beliefs concerning the effect of memory access hops on system performance.

3.4.1 Myth: All Equal Hop Cases Take Equal Time.

As a general rule, any *n* hop case is equivalent to any other *n* hop case in performance, if the only change between the two cases is thread and memory placement. However, there are exceptions to this rule.

The following example demonstrates how a given 1 hop-1 hop case is not equivalent in performance to another 1 hop-1 hop case using the synthetic test. The example shows how saturating the HyperTransport link throughput and stressing the HyperTransport queue buffers can cause this exception to occur.

In the graphs that follow, we compare the following three cases:

- *Threads access local data*

  The first thread runs on node 0 and writes to memory on node 0 (0 hop). The second thread runs on node 1 and writes to memory on node 1 (0 hop).

- *Threads not firing at each other (no crossfire)*

  The first thread runs on node 0 and writes to memory on node 1 (1 hop). The second thread runs on node 1 and writes to memory on node 3 (1 hop).

- Threads firing at each other (*crossfire*)

  The first thread runs on node 0 and writes to memory on node 1 (1 hop). The second thread runs on node 1 and writes to memory on node 0 (1 hop).

In each case, the two threads are run on core 0 of whichever code they are running on. The system is left idle except for the two threads. As shown in Figure 6 on page 26, the crossfire 1 hop-1 hop case is the worst performer.

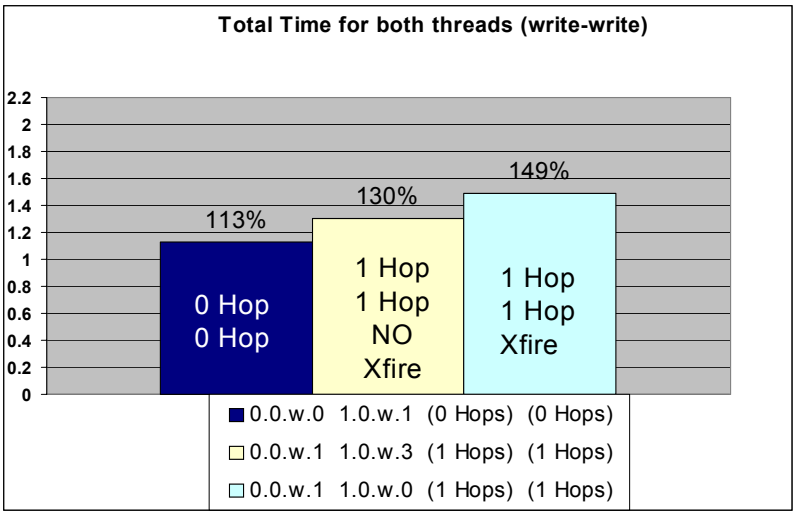

Figure 6. Crossfire 1 Hop-1 Hop Case vs No Crossfire 1 Hop-1 Hop Case on an Idle System

When the write-only threads fire at each other (crossfire), the bidirectional HyperTransport link between node 0 and node 1 is saturated and loaded at 3.5 GB/s in each direction. The theoretical maximum bandwidth of the HyperTransport link is 4 GB/s in each direction. Thus, the utilization of the bidirectional HyperTransport link is 87% (3.5 ÷ 4) in each direction on that HyperTransport link.

On the other hand, when the write-only threads do not fire at each other (no crossfire), the utilization of the bidirectional link from node 0 to node 1 is at 60% in each direction. In addition, the utilization of the bidirectional link from node 1 to node 3 is at 54% in each direction. Since the load is now spread over two bidirectional HyperTransport links instead of one, the performance is better.

The saturation of these coherent HyperTransport links is responsible for the poor performance for the crossfire case compared to the no crossfire case. For detailed analysis, refer to Section A.2 on page 40.

In this synthetic test, read-only threads do not result in poor performance. Throughput of such threads is not high enough to exhaust the HyperTransport link resources. When both threads are read-only, the crossfire case is equivalent in performance to the no crossfire case.

It is also useful to study whether this observation holds on a system that is not idle. The following analysis explores the behavior of the two foreground threads under a variable background load.

Here the same two foreground threads as before were run though the cases as before—*local*, *crossfire* and *no crossfire*. In addition, four background threads are left running on:

- Node 0 (Core 1)
- Node 1 (Core 1)
- Node 2 (Core 0)
- Node 3 (Core 0)

Each of these background threads read a local 64 MB array and the rate of memory demand of each of these threads is varied from low to very high simultaneously. A low rate of memory demand implies that each of the background threads is demanding a memory bandwidth of 0.5 GB/s. A very high rate of memory demand implies that each of the background threads is demanding a memory bandwidth of 4 GB/s as shown in Table 1 on page 16.

Even with the background threads, there are still some free cores left in the system. We call this a *highly subscribed* condition.

This allows us to study the impact of the background load on the foreground threads.

As shown in Figure 7 and Figure 8 on page 28, under both low and very high loads and high subscription, we still observe that the worst performance scenario occurs when write-only threads fire at each other (crossfire).

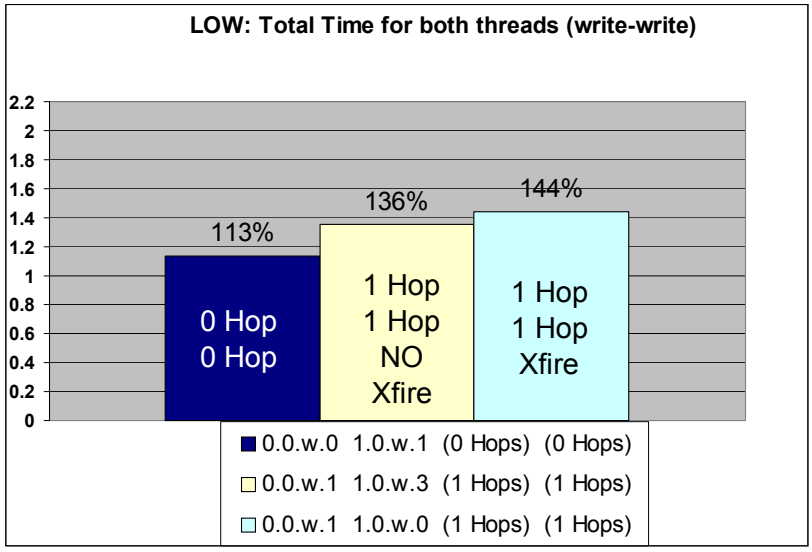

Figure 7. Crossfire 1 Hop-1 Hop Case vs No Crossfire 1 Hop-1 Hop Case under a Low Background Load (High Subscription)

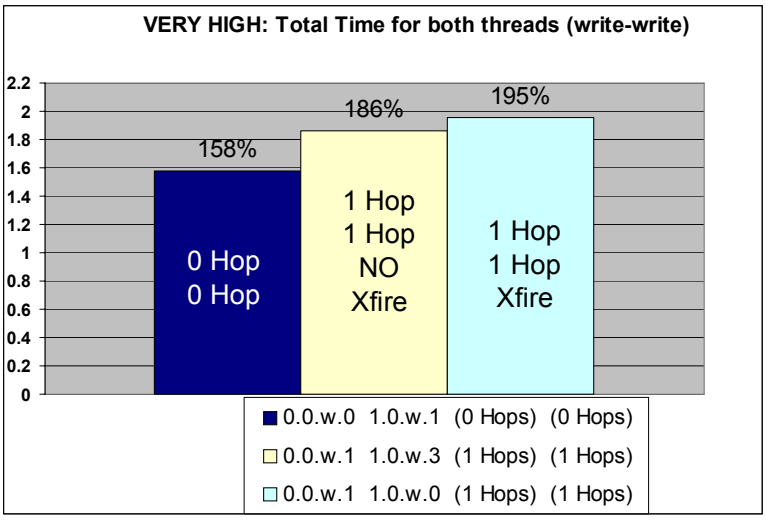

Figure 8. Crossfire 1 Hop-1 Hop Case vs No Crossfire 1 Hop-1 Hop Case under a Very High Background Load (High Subscription)

Next, we increase the number of background threads to six, running on:

- Node 0 (Core 1)
- Node 1 (Core 1)
- Node 2 (Cores 0 and 1)
- Node 3 (Cores 0 and 1)

Each of these background threads reads a local 64 MB array and the rate of memory demand of each thread is very high. A very high rate of memory demand implies that each of the background threads is demanding a memory bandwidth of 4GB/s, as shown in Table 1 on page 16.

No free cores are left in the system. This the *fully subscribed* condition.

As shown in Figure 9 on page 29, when the background load and level subscription are increased to the maximum possible, the no crossfire case becomes slower than the crossfire case.

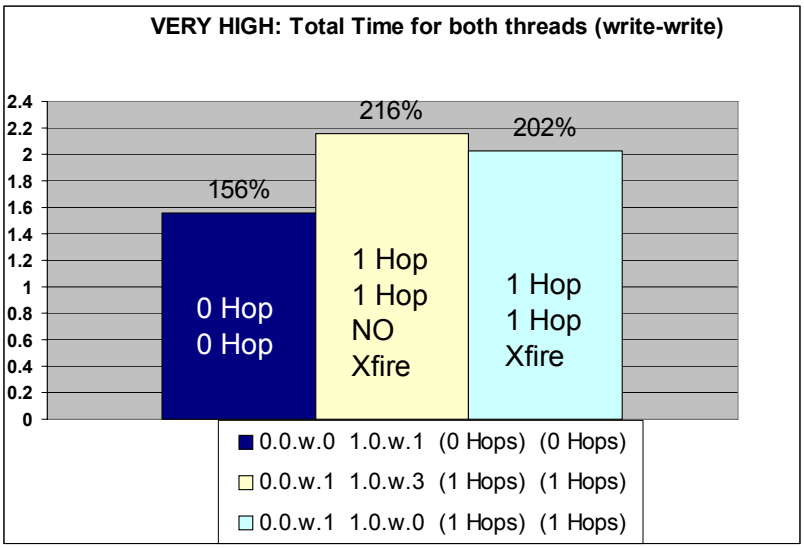

Figure 9. Crossfire 1 Hop-1 Hop Case vs No Crossfire 1 Hop-1 Hop Case under a Very High Background Load (Full Subscription)

In the no crossfire case, the total memory bandwidth observed on the memory controller on node 3 is 4.5 GB/s and several buffer queues on node 3 are saturated. For detailed analysis, refer to Section A.3 on page 42.

Thus, while, in general, all equal hop cases take equal time, there can be exceptions to this rule if some resources in the system—such as HyperTransport link bandwidth and HyperTransport buffer capacity—are saturated

3.4.2 Myth: Greater Hop Distance Always Means Slower Time.

As a general rule, a 2 hop case will be slower than a 1 hop case, which, in turn, will be slower than a 0 hop case, if the only change between the cases is thread and memory placement.

For example, the synthetic test demonstrates how a given 0 hop-0 hop case is slower than a 0 hop-1 hop case. The example shows how saturating memory resources can cause this to occur.

Imagine yourself in the following situation: you are ready to check out at your favorite grocery store with a shopping cart full of groceries. Directly in front of you is a check-out lane with 20 customers standing in line but 50 feet to your left is another check-out lane with only two customers standing in line. Which would you go to? The check-out lane closest to your position has the lowest latency because you don't have far to travel. But the check-out lane 50 feet away has much greater latency because you have to walk 50 feet.

Clearly most people would walk the 50 feet, suffer the latency and arrive at a check-out lane with only two customers instead of 20. Experience tells us that the time waiting to check-out with 20 people ahead is far longer than the time needed to walk to the “remote” check-out lane and wait for only two people.

This analogy clearly communicates the performance effects of queuing time *versus* latency. In a computer server, with many concurrent outstanding memory requests, we would gladly incur some additional latency (walking) to spread memory transactions (check-out processes) across multiple memory controllers (check-out lanes) because this greatly improves performance by reducing the queuing time.

However, if the number of customers at the remote queue increases to 20 or more, then the customer would much rather wait for the local queue directly in front of him.

The following example was extracted by mining the results of the synthetic test case.

There are four cases illustrated in Figure 10. In each case there are two threads running on node 0 (core 0 and core 1 respectively). The system is left idle except for the two threads.

- Both threads access memory on node 0.
- First thread accesses memory on node 0. The second thread accesses memory on node 1, which is one hop away.
- First thread accesses memory on node 0. The second thread accesses memory on node 2, which is one hop away.
- First thread accesses memory on node 0. The second thread accesses memory on node 3, which is two hops away.

As shown in Figure 10, synthetic tests indicate that when both threads are read-only, the 0 hop-0 hop case is faster than the 0 hop-1 hop and 0 hop-2 hop cases.

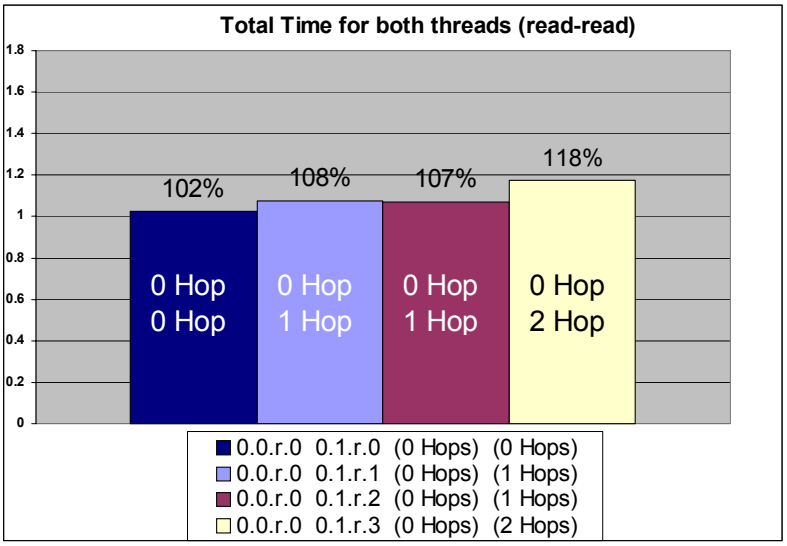

Figure 10. Both Read-Only Threads Running on Node 0 (Different Cores) on an Idle System

However, as shown in Figure 11 on page 31, when both threads are write-only, the 0 hop-1 hop and 0 hop-2 hop cases are faster than the 0 hop-0 hop case.

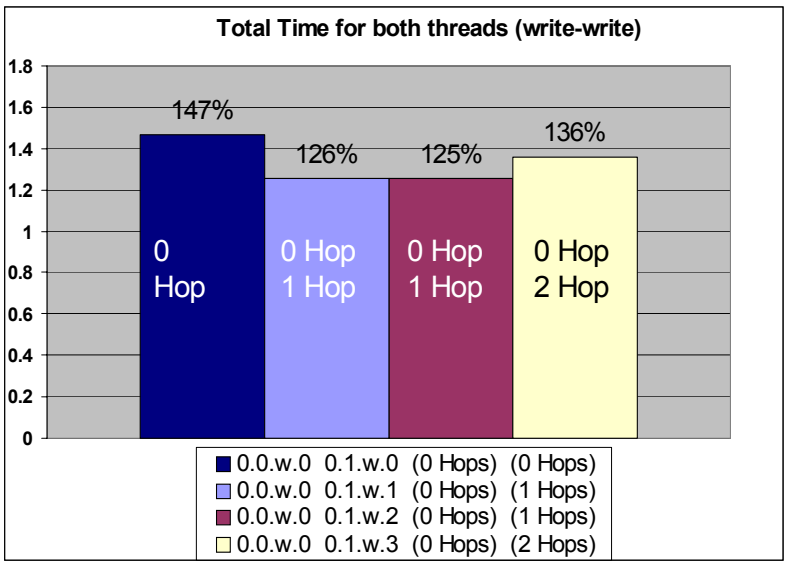

Figure 11. Both Write-Only Threads Running on Node 0 (Different Cores) on an Idle System

When a single thread reads locally, it generates a memory bandwidth load of 1.64 GB/s. Assuming a sustained memory bandwidth of 70% of the theoretical maximum of 6.4 GB/s (PC3200 DDR memory), the cumulative bandwidth demanded by two read-only threads does not exceed the sustained memory bandwidth on that node and hence the local or 0 hop-0 hop case is the fastest.

However, when a single thread writes locally it generates a memory bandwidth load of 2.98 GB/s. This is because each write in this test case results in a cache line eviction and thus generates twice the memory traffic generated by a read. The cumulative memory bandwidth demanded by 2 write-only threads now exceeds the sustained memory bandwidth on that node. The 0 hop-0 hop case now incurs the penalty of saturating the memory bandwidth on that node. For detailed analysis, refer to Section A.4 on page 42.

It is useful to study whether this observation is also applicable under a variable background load.

One would expect that, if the memory bandwidth demanded of the remote node were increased, at some point the 0 hop-1 hop case would become as slow as, and perhaps slower than, the 0 hop-0 hop case for the write-only threads.

The same two write-only threads as before are running on node 0, going though the following cases:

- Both threads access local memory.
- First thread accesses local memory and second thread accesses memory that is remote by one hop.
- First thread accesses local memory and second thread access memory that is remote by two hops.

In addition, three background threads are running on nodes 1, 2 and 3.

Each of these background threads access data locally. The rate of memory demand by each these threads is varied simultaneously from low to medium to high to very high as shown in Table 1 on page 16. This allows us to study the impact of the background load on the foreground threads and evaluate the performance of the two foreground threads.

Even with the background threads, there are still some free cores left in the system; this is called the *highly subscribed condition*.

As shown in Figures12, 13 below and Figure 14 on page 33, as the load increases from low to medium to high, the advantage of having the memory for one of the writer threads one or two hops away diminishes.

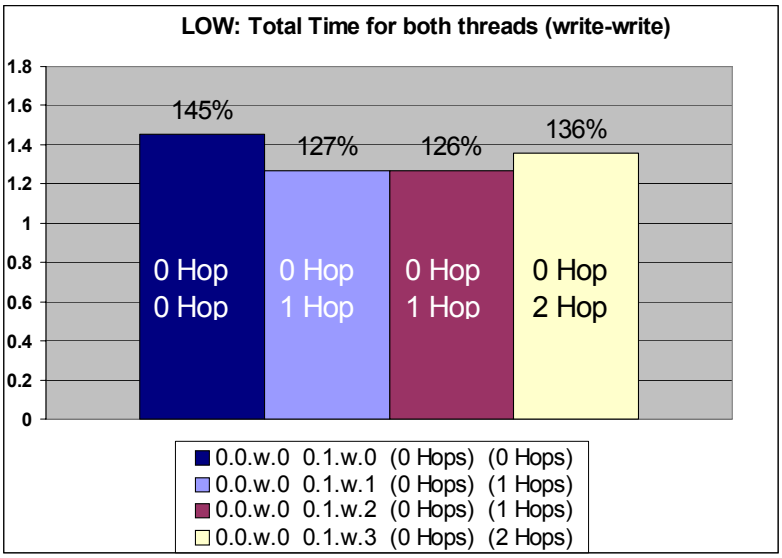

Figure 12. Both Write-Only Threads Running on Node 0 (Different Cores) under Low Background Load (High Subscription)

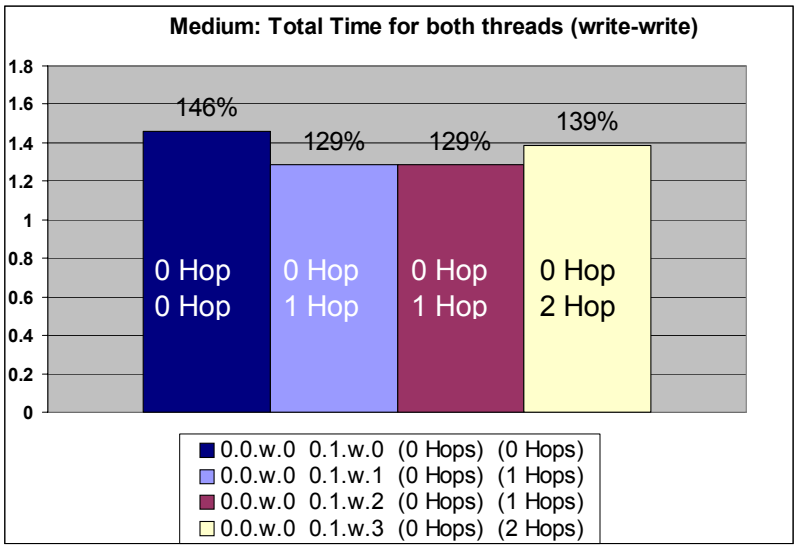

Figure 13. Both Write-Only Threads Running on Node 0 (Different Cores) under Medium Background Load (High Subscription)

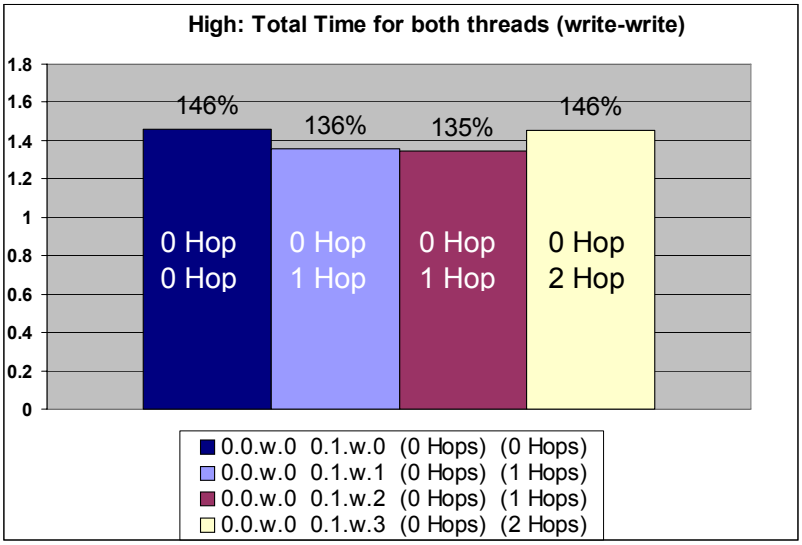

Figure 14. Both Write-Only Threads Running on Node 0 (Different Cores) under High Background Load (High Subscription)

As shown in Figure 15 below, the advantage disappears at very high loads and the 0 hop-1 hop case becomes slower than the 0 hop-0 hop case.

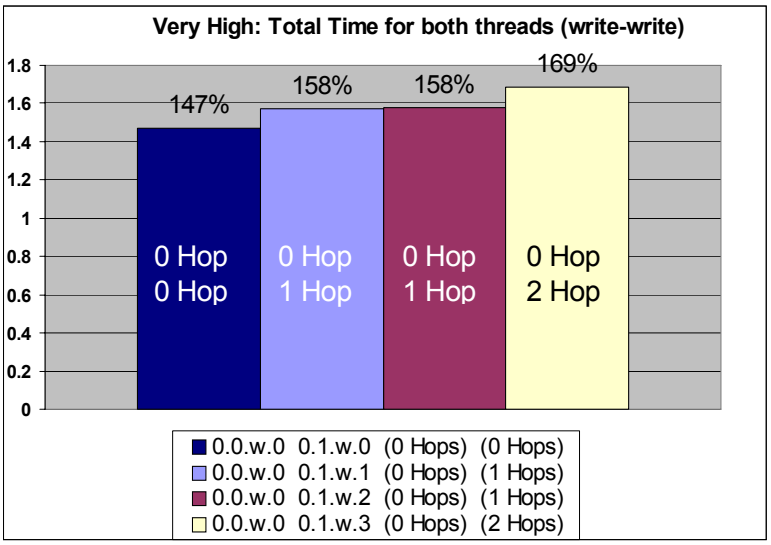

Figure 15. Both Write-Only Threads Running on Node 0 (Different Cores) under Very High Background Load (High Subscription)

Under a very high background load, for the 0 hop-1 hop case, there is a total memory access rate of 4.78 GB/s on node 1. Several buffer queues on node 1 are saturated. For detailed analysis, refer to section Section A.5 on page 43.

Thus, greater hop distance does not always mean slower time. Remember that it is still advised that the developer keep the data local as much as possible. In the analogy used above, if the local queue has 20 customers and the remote one has two, the customer would much rather have been standing in front of the queue with two customers and make that his local queue in the first place. In the synthetic case above, keeping the first thread on node 0 doing local writes and the second thread on node 1 doing local writes would be the fastest.

3.5 Locks

In general, it is good practice for user-level and kernel-level code to keep locks aligned to their natural boundaries. In some hardware implementations, locks that are not naturally aligned are handled with the mechanisms used for legacy memory mapped I/O and should absolutely be avoided if possible.

If a lock is aligned properly, it is treated as a faster cache lock. The significantly slower alternative to a cache lock is a bus lock, which should be avoided at all costs. Bus locks are very slow and force serialization of many operations unrelated to the lock within the processor. Furthermore bus locks prevent the entire HyperTransport fabric from making forward progress until the bus lock completes. Cache locks on the other hand are guaranteed atomicity by using the underlying cache coherence of the ccNUMA system and are much faster.

3.6 Parallelism Exposed by Compilers on AMD ccNUMA Multiprocessor Systems

Several compilers for AMD multiprocessor systems provide additional hooks to allow automatic parallelization of otherwise serial programs. Several compilers also support the OpenMP API for parallel programming. For details about support for auto parallelization and OpenMP in various compilers, see the references [4], [14], [15] and [16].

Chapter 4 Conclusions

The single most important recommendation for most applications is to keep data local on node where it is being accessed. As long as a thread initializes the data it needs, in other words writes to it for the first time, a ccNUMA aware OS will typically keep the data local on the node where the thread runs. By allowing local allocation on first touch policy for data placement, a ccNUMA aware OS makes the task of data placement transparent and easy for most developers.

In some cases, if an application demonstrates symptoms that its threads are being moved away from their data, it might be useful to explicitly pin the thread to a specific node. Several ccNUMA-aware OSs offer tools and APIs to influence thread placement. Typically an OS scheduler uses load balancing schemes to make decisions on where to place threads. Using these tools or APIs will override the scheduler and hand over the control for thread placement to the developer. . The developer should do reasonable scheduling with these tools or APIs by adhering to the following guidelines:

- When scheduling threads that mostly access independent data on an idle dual-core AMD multiprocessor system, first schedule threads to an idle core of each node until all nodes are exhausted and then move on to the other idle core of each node. In other words, schedule using node major order first followed by core major order.
- When scheduling multiple threads that mostly share data with each other on an idle dual-core AMD multiprocessor system, schedule threads on both cores of an idle node first and then move on to the next idle node and so on. In other words, schedule using core major order first followed by node major order.

By default, a ccNUMA-aware OS uses the local allocation on first touch policy for data placement. Normally practical, this policy can become suboptimal if a thread first touches data on one node that it subsequently no longer needs, but which some other thread later accesses from a different node. It is best to change the data initialization scheme so that a thread initializes the data it needs and does not rely on any other thread to do the initialization. Several ccNUMA aware OSs offer tools and APIs to influence data placement. Using the tools or API will override the default local allocation by first touch policy and hand over the control for data placement to the developer.

If it is not possible to change the data initialization scheme or if the data is truly shared by threads running on different nodes, then a technique called *node interleaving* of memory can be used. The use of node interleaving on data is recommended when the data resides on a single node and is accessed by three or more cores. The interleaving should be performed on the nodes from which the data is accessed and should only be used when the data accessed is significantly larger than 4K (when the system is configured for normal pages, which is the default) or 2M (when the system is configured for large pages). Developers are advised to experiment with their applications to gauge the performance change due to node interleaving. For additional details on the tools and APIs offered by various OS for node interleaving, refer to Section A.8 on page 46.

Data placement tools can also come in handy when a thread needs more data than the amount of physical memory available on a node. Certain OSs also allow data migration with these tools or API. Using this feature, data can be migrated from the node where it was first touched to the node where it is subsequently accessed. There is a cost associated with this migration and it is not advised to use it frequently. For additional details on the tools and APIs offered by various OS for thread and memory placement refer to Section A.7 on page 44.

It is recommended to avoid sharing of data resident within a single cache line between threads running on different cores.

Advanced developers may also run into interesting cases when experimenting with the thread and data placement tools and APIs. Sometimes, when comparing workloads that are symmetrical in all respects except for the thread and data placement used, the expected symmetry may be obscured. These cases can mostly be explained by understanding the underlying system and avoiding saturation of resources due to an imbalanced load.

The buffer queues constitute one such resource. The lengths of these queues are configured by the BIOS with some hardware-specific limits that are specified in the BIOS Kernel and Developers Guide for the particular processor. Following AMD recommendations, the BIOS allocates these buffers on a link-by-link basis to optimize for the most common workloads.

In general, certain pathological access patterns should be avoided: several nodes trying to access data on one node or the crossfire scenario can saturate underlying resources such as the HyperTransport™ link bandwidth and HyperTransport buffer queues and should be avoided when possible. AMD makes event profiling tools available that developers can use to analyze whether their application is demonstrating such behavior.

AMD very strongly recommends keeping user-level and kernel-level locks aligned to their natural boundaries.

Some compilers for AMD multiprocessor systems provide additional hooks to allow for automatic parallelization of otherwise serial programs. There is also support for extensions to the OpenMP directives that can be used by OpenMP programs to improve performance.

While all the previous conclusions are stated in the context of threads, they can also be applied to processes.

Appendix A

The following sections provide additional explanatory information on topics discussed in the previous sections of this document.

A.1 Description of the Buffer Queues

Figure 16 shows the internal resources in each Quartet node. The memory controller (MCT), the System Request Interface (SRI) and the Crossbar (XBar) on each node all have internal buffers that are used to queue transmitted transaction packets. Physically each of these units only has two kinds of buffers, command and data.

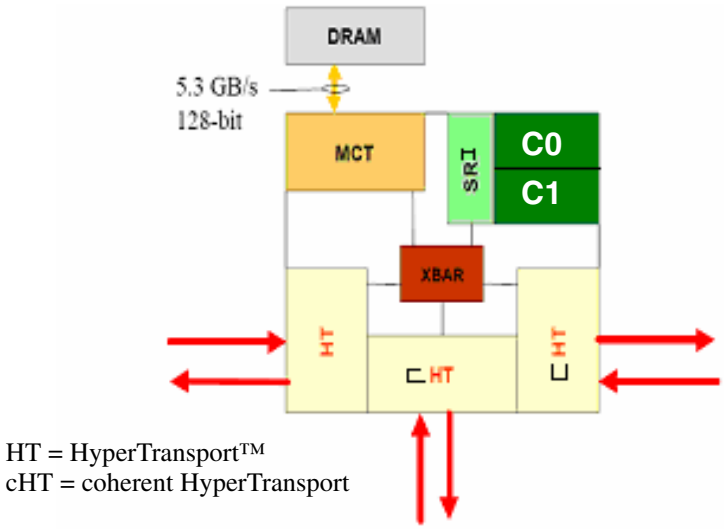

Figure 16. Internal Resources Associated with a Quartet Node

Consider the buffers that come into play in the interface between the XBar and the HyperTransport™ links.

Each node has an incoming and outgoing coherent HyperTransport link on its XBar to every other node but one in the system. Node 0 has an outgoing link that allows it to send data from node 0 to node 1. Likewise node 0 has an incoming link from node 1 to node 0. The two links together can be considered as one bidirectional link.

Each node has HyperTransport buffers in its XBar that are used to queue up the packets that are going to be sent on the outgoing link. The sending node does not send the packets until the receiving node is ready to receive them.

Now Consider the buffers that come into play in the interface between the XBar and the MCT. Packets to be transmitted from the XBar to the MCT are queued in the ‘XBar-to-MCT” buffers.

Likewise packets to be transmitted from the MCT to the XBar are queued in the "MCT-to-XBar" buffers. The buffers in the SRI, XBar and MCT can be viewed as staggered queues on the various units.

A.2 Why Is the Crossfire Case Slower Than the No Crossfire Case on an Idle System?

The following analysis highlights some of the important characteristics of the underlying resources that come into play when there is crossfire *versus* no crossfire.

A.2.1 What Resources Are Used When a Single Read-Only or Write-Only Thread Accesses Remote Data?

When a thread running on node 0 reads data from node 1, on an otherwise idle system, there is traffic on both the incoming and outgoing links.

When a node makes a read memory request from a memory controller, it first sends a request for the memory to that memory controller, which can be local or remote. That memory controller then sends probes to all other nodes in the system to see if they have the memory in their cache. Once it receives the response from the nodes, it sends a response to the requesting node. Finally it also sends the read data to the requesting node.

When a thread running on node 0 reads data from node 1, it sees non-data traffic (loaded at 752 MB/s) on the outgoing link and both data and non-data traffic on the incoming link (2.2 GB/s). There is also some non-data traffic on the coherent HyperTransport links that connect nodes other than nodes 0 and 1 because of the probes and the responses.

When a thread running on node 0 writes data to node 1, it sees as much data traffic on the incoming link as it does on the outgoing link (incoming and outgoing link each at 2.2 GB/s). In this synthetic test case, there are several successive writes happening to successive cache line elements of a 64MB array. These result in steady state condition of a cache line eviction or write back for each write access. Each write access from node 0 to node 1 triggers a data read from node 1 and then a data write to node 1.

A.2.2 What Resources Are Used When Two Write-only Threads Fire at Each Other (Crossfire) on an Idle System?

Assuming the coherent HyperTransport links between node 0 and node 1 have infinite throughput capacity, it is expected that, when the write-only threads fire at each other, the throughput on each of these links would be twice that observed when a single write-only thread running on node 0 is writing to node 1, i.e., 2*(2.2 GB/s).

The theoretical maximum HyperTransport bandwidth of each coherent HyperTransport link between node 0 and node 1 is at 4 GB/s. Hence we can not expect the HyperTransport bandwidth to reach the

4.4 GB/s necessary. The two coherent HyperTransport links are loaded at 3.5 GB/s each. Thus the utilization of each of the two coherent HyperTransport links that connect node 0 and node 1 equals 87% (3.5÷4).

A.2.3 What Role Do Buffers Play in the Throughput Observed?

Node 0 queues up packets in HyperTransport buffers and sends them on the outgoing link only if node 1 can accommodate them. Likewise node 1 queues up packets in HyperTransport buffers and sends them on the outgoing link only if node 0 can accept them.

When the HyperTransport buffers are saturated, they can prevent the coherent HyperTransport links from reaching their full throughput capacity of 4GB/s and, thus, full 100% utilization.

Also, saturating the HyperTransport buffers in the XBar has a domino effect on the other buffers in the system. Remember, the SRI is connected to the XBar, which is connected to the coherent HyperTransport links.

When packets are stalled in the XBar buffer queue to be sent over the coherent HyperTransport links, a chain effect can cause packets stall in the SRI buffer queue to be sent to the XBar.

AMD makes several event profiling tools available under NDA to monitor the HyperTransport bandwidth and buffer queue usage patterns.

The buffer lengths are BIOS configurable within some hardware-specific limits that are specified in the appropriate BIOS Kernel and Developers Guide for the processor under consideration. Following AMD recommendations, the BIOS allocates these buffers on a link-by-link basis to optimize for the most common workloads.

A.2.4 What Resources Are Used When Write-Only Threads Do Not Fire at Each Other (No Crossfire) on an Idle System?

Now consider the case in which the writer threads do not fire at each other: i.e., the first thread runs on node 0 and writes to memory on node 1 and second thread runs on node 1 and writes to memory on node 3.

In this case, the bidirectional link from node 0 to node 1 is in under substantial use (60% utilization in each direction). In addition, the bidirectional link from node 1 to node 3 is also under substantial use (54% utilization in each direction).

As the load is now spread over two bidirectional links instead of 1, the performance is better than in the crossfire case.

A.3 Why Is the No Crossfire Case Slower Than the Crossfire Case on a System under a Very High Background Load (Full Subscription)?

When the threads are firing at each other (crossfire) and all other free cores are running background threads at very high load, the system sees the following traffic pattern, where each node receives memory requests from the threads as described:

- Node 0: 1 background and 1 foreground threads.
- Node 1: 1 background and 1 foreground threads.
- Node 3: 2 background threads.
- Node 2: 2 background threads.

In the no crossfire case, the system sees the following traffic pattern:

- Node 0: 1 background thread
- Node 1: 1 background and 1 foreground threads.
- Node 3: 2 background and 1 foreground threads.
- Node 2: 2 background threads.

The no crossfire case suffers from a greater load imbalance than the crossfire case with node 3 suffering the worst effect of this imbalance.

Remember that each of the background threads asks for data at a rate of 4GB/s and each of the foreground threads asks for data at a rate of 2.98 GB/s.

Data shows that there is total memory access of 4.5GB/s on node 3 and that several buffer queues on node 3 are saturated and cannot absorb the data provided by the memory controller any faster.

A.4 Why Is 0 Hop-0 Hop Case Slower Than the 0 Hop-1 Hop Case on an Idle System for Write-Only Threads?

When both write-only threads running on different cores of node 0 access data locally (0 hop-0 hop), significant demands are placed on the local memory on node 0.

Data demonstrates that there is total memory access of 4.5 GB/s on node 0. The memory on node 0 cannot handle requests for data any faster and is running at full capacity. Several buffer queues on node 0 are saturated and waiting for the memory requests to be serviced.

A.5 Why Is 0 Hop-1 Hop Case Slower Than 0 Hop-0 Hop Case on a System under High Background Load (High Subscription) for Write-Only Threads?

When a 0 hop-0 hop scenario is subjected to a very high background load, the system sees the following traffic pattern, where each node gets memory requests from the threads as described:

- Node 0: 2 foreground threads.
- Node 1: 1 background thread.
- Node 3: 1 background thread.
- Node 2: 1 background thread.

In the 0 hop-1 hop case, the system sees the following traffic pattern:

- Node 0: 1 foreground thread
- Node 1: 1 foreground and 1 background threads.
- Node 3: 1 background thread.
- Node 2: 1 background thread.

The 0 hop-1 hop case suffers from a greater load imbalance than the 0 hop-0 hop case, with node 1 suffering the worst effect of this imbalance.

Each of the background threads, as before, asks for data at a rate of 4GB/s and each of the foreground threads asks for data at a rate of 2.98 GB/s.

Data shows that there is a total memory access rate of 4.78 GB/s on node 1 and several buffer queues on node 1 are saturated and cannot absorb the data provided by the memory controller any faster.

A.6 Support for a ccNUMA-Aware Scheduler for AMD64 ccNUMA Multiprocessor Systems

Developers should ensure that the OS is properly configured to support ccNUMA. All versions of Microsoft® Windows® XP for AMD64 and Windows Server for AMD64 support ccNUMA without any configuration changes. The 32-bit versions of Windows Server 2003, Enterprise Edition and Windows Server 2003, Datacenter Edition require the /PAE boot parameter to support ccNUMA. For 64-bit Linux®, there may be separate kernels supporting ccNUMA that should be selected. The 2.6.x Linux kernels feature NUMA awareness in the scheduler[11]. Most SuSE and Red Hat Enterprise distributions of 64-bit Linux have the ccNUMA aware kernel. Solaris 10 and subsequent versions of Solaris for AMD64 support ccNUMA without any changes.

A.7 Tools and APIs for Thread/Process and Memory Placement (Affinity) for AMD64 ccNUMA Multiprocessor Systems

This following sections discuss tools and APIs available for assigning thread/process and memory affinity under various operating systems.

A.7.1 Support Under Linux®

Linux provides command-line utilities to explicitly set process/thread and meory affinity to both nodes and cores on a node[5]. Additionally, **libnuma**, a shared library, is provided for more precise affinity control from within applications.

Controlling Process and Thread Affinity

The Linux command-line utlities offer high-level affinity control options. The **numactl** utility is a command line tool for running a process with a specific *node* affinity.

For example, to run the `foobar` program on the cores of node 0, enter the following at the command prompt:

```
numactl --cpubind=0  foobar
```

Application and kernel developers can use the **libnuma** shared library, which can be linked to programs and offers a stable API for setting thread affinity to a given node or set of nodes. Interested developers should consult the Linux **man** pages for details on the various functions available.

On a dual-core processor, a node has more than one core. If a process or thread is affined to a particular node using the tools or API discussed above, it may still migrate back and forth between the two cores of that node. This migration may or may not affect performance.

The **taskset** utility is a command-line tool for setting the process affinity for a specified program to any core. For example, to run program `foobar` on the two cores of node 0, enter the following on the command line:

```
taskset -c 0,1  foobar
```

In SuSE Linux Enterprise Server 10/10.1, the **numactl** utility can be used instead of **taskset** to set process affinity to any core.

Linux provides several functions by which to set the thread affinity to any core or set of cores:

- **pthread_attr_setaffinity_np( )** and **pthread_create( )** are provided as a part of the older **nptl** library; they can be used to set the affinity parameter and then create a thread using that affinity.
- **sched_setaffinity( )** system call and **schedutils** scheduler utilities.

Controlling Memory Affinity

Both **numactl** and **libnuma** library functions can be used to set memory affinity[5]. Memory affinity set by tools like **numactl** applies to all the data accessed by the entire program (including child processes). Memory affinity set by **libnuma** or other library functions can be made to apply only to specific data as determined by the program.

Both **numactl** and the **libnuma** API can be used to set a preferred memory affinity instead of forcibly binding it. In this case the binding specified is a hint to the OS; the OS may choose not to adhere to it.

At a high level, normal first touch binding, explicit binding and preferred binding are all available as memory policies on Linux.

By default, when none of the tools/API is used, Linux uses the first touch binding policy for all data. Once memory is bound, either by the OS, or by using the tools/API, the memory will normally remain resident on that node for its lifetime.

A.7.2 Support under Solaris

Sun Solaris provides several tools and API's for influencing thread/process and memory affinity[6].

Solaris provides a command line tool called **pbind** to set process affinity. There is also a shared library called **liblgrp** that provides an API that a program can call to set thread affinity.

Solaris provides a memory placement API to affect memory placement. A program can call the **madvise( )** function to provide hints to the OS as to the memory policy to use. This API does not allow binding of memory to an explicit node or set of nodes specified on the command line or in the program. But there are several policies other than the first touch policy that can be used.

For example, a thread can use **madvise** to migrate the data it needs to the node where it runs, instead of leaving it on a different node, on which it was first touched by another thread. There is, naturally, a cost associated with the migration.

Solaris provides a library called **madv.so.1** which can interpose on memory allocation system calls and call the madvise function internally for the memory policy.

By default, Solaris uses the first touch binding policy for data that is not shared. Once memory is bound to a node it normally remains resident on that node for its lifetime.

Sun is also working on supporting several command line tools to control thread and memory placement. These are expected to be integrated in the upcoming versions of Solaris, but experimental versions are currently available[7].

A.7.3 Support under Microsoft® Windows®

In the Microsoft Windows environment, the function to bind a thread on particular core or cores is **SetThreadAffinityMask( )**. The function to run all threads in a process on particular core or cores is **SetProcessAffinityMask( )**[8].

The function to set memory affinity for a thread is **VirtualAlloc( )**[9]. This function gives the developer the choice to bind memory immediately on allocation or to defer binding until first touch.

Although there are no command-line tools for thread/process and memory placement, several Microsoft Enterprise products provide NUMA support and configurability, such as SQL Server 2005 [10] and IIS [11].

If an application relies on heaps in Windows, we recommend using a low fragmentation heap (LFH) and using a local heap instead of a global heap[12][13].

By default, Windows uses the first touch binding policy for all data. Once memory is bound to a node, it normally resides on that node for its lifetime.

A.8 Tools and APIs for Node Interleaving in Various OSs for AMD64 ccNUMA Multiprocessor Systems

This section discusses tools and APIs available for performing node interleaving under various operating systems.

A.8.1 Support under Linux®

Linux provides several ways for an application to use node interleaving [5].

- **numactl** is a command line tool, which is used for node interleaving all memory accessed by a program across a set of chosen nodes.

For example, to interleave all memory accessed by program `foobar` on nodes 0 and 1, use:

```
numactl --interleave=0x03  foobar
```

- **libnuma** offers several functions a program can use to interleave a given memory region across a set of chosen nodes.

Linux only supports the round robin node interleaving policy.

A.8.2 Support under Solaris

Solaris offers an API called **madvise**, which can be usedwith the **MADV_ACCESS_MANY** flag to tell the OS to use a memory policy that causes the OS to bind memory randomly across the nodes. This offers behavior similar to the round robin node interleaving of memory offered by Linux.

This random policy is the default memory placement policy used by Solaris for shared memory.

A.8.3 Support under Microsoft® Windows®

Microsoft Windows does not offer node interleaving.

A.8.4 Node Interleaving Configuration in the BIOS

AMD Opteron™ and Athlon™ 64 ccNUMA multiprocessor systems can be configured in the BIOS to interleave all memory across all nodes on a page basis (4KB for regular pages and 2M for large pages). Enabling node interleaving in the BIOS overrides the use of any tools and causes the OS to interleave all memory available to the system across all nodes in a round robin manner.